– Independent Work Report Spring 2020 –

Multi-dimensional Rewards are Unnecessary in Deterministic Environments

Nimra Nadeem
Adviser: Tom Griffiths

Abstract

This paper compares the effect of single-reward v. multi-reward functions on objective fitness functions in a fully deterministic reinforcement learning environment. The experiments are conducted for two setups: (1) binary rewards with 6 items and (2) a discrete range of rewards with 2 items. The paper aims to take the first step in building a computational model of animals with numerous sources of reward in natural environments.

1. Introduction

Mindfulness and yogic practices rooted in Buddhist wisdom are being increasingly incorporated into contemporary medicine practices. [14] Specifically in modern psychology, these practices have proven constructive as part of treatment protocols. [28] Dialectical Behavior Therapy (DBT), a psychotherapy technique which has Mindfulness as one of its 4 essential pillars, has been widely studied and is used as an effective treatment for borderline personality disorder, depressive disorder and anxiety disorders. [13, 18, 11]

These mindfulness techniques focus on reducing emotional responses to experiences that cause pleasure and pain. [28] The fundamental Buddhist teaching informing this technique focuses on eliminating craving, in other words, eliminating desires. [2] Studies support this approach by showing that the more desires one has, the unhappier one becomes. [19, 24]

On the other hand, human actions are motivated by the very desires that these mindfulness techniques attempt to lessen. [12, 21] If our happiness and mental health is elevated in response to the reduction of emotional responses, i.e. desires, then why have humans developed so many desires in the first place? Why have we evolved in a way that our actions are driven by our motivations, which in turn are controlled by the numerous desires/cravings/rewards we experience?

The computational framework I use to explore this puzzle assumes the following. The evolutionary process is driven by the need to maximize fitness, not by the need to maximize individual happiness. [9, 16] As such there are two mechanisms at play: (1) an individual's goal to maximize happiness, and (2) nature's goal to maximize fitness. The individual's reward system is a tool for achieving the latter goal of maximizing fitness.

The goal of my research is to test the optimality of multiple rewards in various artificial environments with a given fitness function. I conduct these tests using a computational reinforcement learning framework, where an agent's aim is to maximize reward using an ε-greedy Q-learning algorithm. In this paper, I report my results for several deterministic environments. My major finding is that in deterministic environments, multiple rewards are unnecessary. My future goal is to conduct similar tests in stochastic environments.

My results can give insight into the kinds of artificial environments that give rise to naturalistic animal reward processes. In addition, they can further our understanding of optimal reward functions for specific computational reinforcement learning tasks.

2. Problem Background and Related Work

Motivation can be loosely defined as the cause of actions in autonomous agents. Studies in psychology and neuroscience have established that both intrinsic and extrinsic motivation are controlled by the experience of internal or external rewards. [23, 12, 15, 22, 5] [1] The reward pathway of the human brain causes the subjective experience of "rewards" in individuals, primarily by affecting the release of the neurotransmitter dopamine. [8, 5] Some works in philosophy and psychology have theorized about *why* there are so many different sources of reward. [17, 3] However an empirical framework for answering this question has not yet been suggested.

Computational Reinforcement Learning (RL) models include an environment (represented as

[1] Intrinsic motivation is the desire to do something for the pleasure the action itself brings. Extrinsic motivation is the desire to do something that acts as a means to an end where the end is the source of pleasure. [21]

a set of states), an agent, a set of actions that the agent may perform on the environment, and a reward function that associates each state-action pair with a numeric value. The agent's goal is to maximize the total reward over the course of its lifetime. [26, 1] Computational models of human learning have been built using this reinforcement learning framework. [27, 10, 6, 23] Such models use the reward functions in a reinforcement learning problem as a representation of primary rewards in animal reward processes. [25, 20, 10] However, the classic RL framework requires the reward function as an input parameter, which is known a priori. The purpose of my research is to discover the parameters controlling the reward function itself.

Some previous work has attempted to modify the classic RL model to incorporate learning systems where the reward function is not a known parameter. [25, 6, 27] This line of work has used such modified models to simulate evolutionary processes, but none of the studies has aimed to specifically explore the question of multiple rewards. I will use a modified RL model developed by Singh, Lewis and Barto (2009) as an empirical framework for my research question. [25]

3. Approach

In a work by Singh, Lewis and Barto (2009) a modified version of the classic RL model was suggested. [25] In this modified model, the agent acts on the external environment based on the decision it makes. The external environment responds with feedback in the form of "sensations". The sensations in turn act on the internal environment of the agent to produce internal rewards. The experience of these rewards informs the future decisions the agent makes.

Figure (1) shows the modified reinforcement learning model I use to represent the interdependence of an agent's (individual's) goal to maximize happiness, versus the designer's (nature's) goal to maximize "fitness". The crucial difference between the aims of the work by Singh et al. (2009) and my study is the space of environments and the space of reward functions that we aim to explore.

The same paper by Singh et al. (2009) proposes an algorithm for computing the *optimal reward function* for an environment which maximizes a given *fitness function*. [25] At first glance it might seem that fitness–based reward functions (i.e. greater fitness $\rightarrow$ greater reward) might be the best

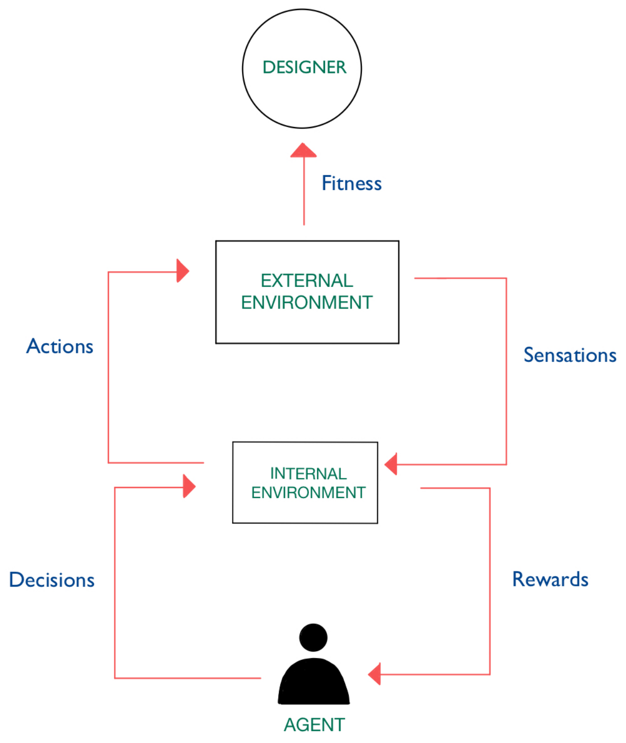


Figure 1: The diagram above describes the modified Reinforcement Learning model that I use. The agent makes decisions which act as signals to the agent's internal environment. This results in the agent's actions on the external environment. These actions result in 2 things: (1) sensations for the agent and (2) fitness value for the designer. The sensations act on the internal environment of the agent to produce rewards. The experience of rewards informs the decision the agent makes. This model is adapted from Barto et al. (2004). [7]

solution. However, according to the findings in this paper, fitness-based reward function are *not* always the fitness-maximizing reward function. [25]

Thus, I use fitness as a measure of the effectiveness of a given reward function. Using the optimal reward algorithm, I run my experiments over all possible reward functions. I then compare the fitness value of single-reward functions with the fitness value of multi-reward functions.

The classic reinforcement learning framework requires multiple episodes, with a specific number of steps each, for the agent to learn the reward-maximizing policy. In contrast, to represent an animal's "lifetime", I use only a single episode with 10,000 steps. For this reason, my agent uses an ε-greedy Q-learning algorithm. Unlike other Temporal Difference Learning algorithms, Q-learning uses online learning, i.e. the information gained at each step is used to update all recently visited states. [1]

4. Implementation

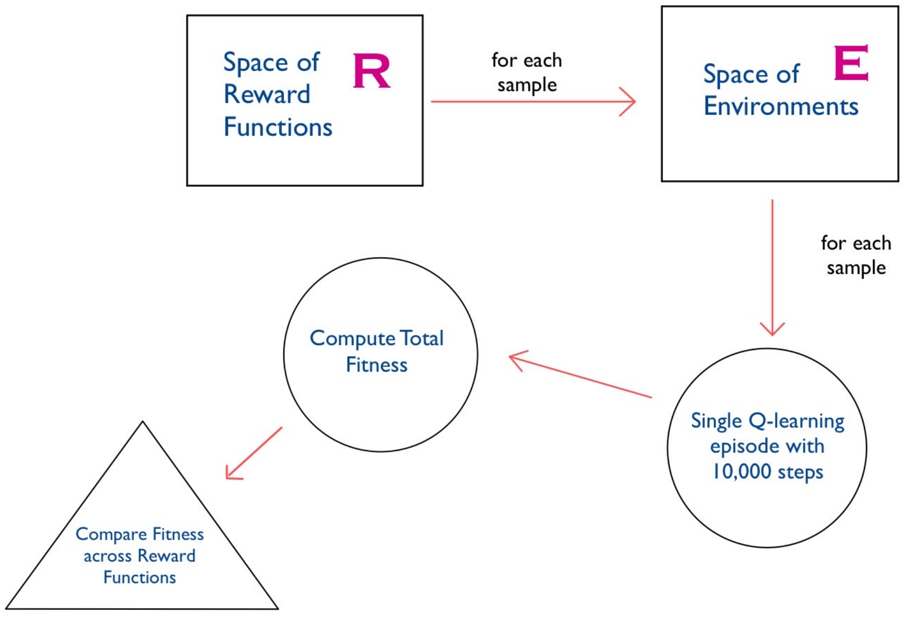

Space of Reward Functions
R
for each sample
Space of Environments
E
for each sample
Single Q-learning episode with 10,000 steps
Compute Total Fitness
Compare Fitness across Reward Functions

Figure 2: The overall workflow of my implementation. I pick each element of the set of reward functions in turn. For each of these elements, I randomly sample 100 environments. For each sampled environment, I run a single ε-greedy Q-learning episode with 10,000 steps and no terminal state. During this episode, the agent's state determines the increase or decrease in the overall fitness value. I then compute the average total fitness across all sampled environments for the selected reward function.

4.1. Experiment Design

For the implementation, I adapted the work done by Rachit Dubey on an open source code base available on GitHub. [4] I use the optimal reward algorithm to determine the optimal reward function for two different setups.

1. A deterministic environment with binary rewards
2. A deterministic environment with a discrete range of rewards

For both setups, the physical space is represented as a 10 x 10 grid. The set of states, S, consists of the 100 unique locations on this grid. There are four possible actions: **Up**, **Down**, **Left**, **Right**. The agent lives for a total of 10,000 steps in each experiment, i.e. there is a single "lifetime" which is not divided into multiple episodes.

For both setups, I ran the entire experiment twice. Once with the initial state randomly set in the central region of the grid. The second time with the initial state randomly set anywhere on the grid. The results for both setups were similar across the two initial states. Thus, for this paper, I include analysis just for the first case in both setups, i.e. the initial state randomly set in the central region of the grid.

Setup 1: Deterministic Environment with Binary Rewards

In the first setup, There are 6 'food items', labelled x, y, z, q, w, e. Each food item has a designated column, and 2 possible rows it can occupy. The row for a specific environment sample is chosen randomly from the two options. For example, y is always in the third column, and is randomly chosen to be either in the 2nd or 3rd row each time an environment is sampled. The 6 items are located in the top left, top middle, top right, bottom left, bottom middle, bottom right regions of the grid respectively. The column and row specifications are set in order to ensure a somewhat even dispersal of food items over the physical space.

The fitness value of an episode represents the number of calories consumed in the lifetime. Each time the agent steps into a grid location with a food item, the fitness value is increased by 1. Each time the agent steps into a grid location without a food item, the fitness value is decreased by 0.1. The total fitness value is used to evaluate each reward function.

The reward functions in this setup only includes binary possibilities for each food item. That is, each food item can either have a reward value of 0 (dislike) or 1 (like).

My implementation of the optimal reward algorithm loops through each possible combination of values for the 6 items. This gives a total of $2^6 = 64$ unique reward functions.

Setup 2: Deterministic Environment with a Discrete Range of Rewards

In the second setup, there are 2 'food items', labelled y and e. Similar to the first setup, each food item has a designated column, and 2 possible rows it can occupy. The 2 food items are located in the top left and bottom right regions of the grid respectively.

The fitness function is the same as setup 1. The reward functions in this setup include a discrete range of possibilities for each food item. Each food item can take any integer reward value in the range $[-3,4]$ with the upper bound exclusive. My implementation of the optimal reward algorithm loops through each possible combination of values for the 2 items. This gives a total of $7^2 = 49$ unique reward functions.

4.2. Algorithm

For both setups, each possible reward function is evaluated by the fitness value over an agent's lifetime in an environment with the given reward function. The space of reward functions, R, includes combinations of single and multiple reward functions. The space of environments, E, includes the grid worlds with varying location of food items, rewards and initial state. The agent uses an ε-greedy Q-learning algorithm to learn the optimal policy during its lifetime. Since our experiment does not rely on multiple episodes, rather a single prolonged lifetime, the Q-learning algorithm is particularly suitable.

The update rule used for the Q-learning algorithm [1] is

$$U(s_t) \leftarrow U(s_t) + \alpha[r_{t+1} + \gamma U(s_{t+1}) - U(s_t)]\ e_t(s) \qquad \forall s \in S \tag{1}$$

Where $U(s_t)$ is the utility of state s at time step t and $e_t(s)$ is the eligibility trace for each state at time step t, calculated as follows

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & s = s_t \end{cases}$$

The parameters used in the Q-learning algorithm are the same across both set ups.

- Q_0, the initial utility of each state. I use a value of 10.
- γ, the discount rate, which accounts for the agent valuing the immediate reward more than the potential future rewards. I use a value of 0.99.

- α, the learning rate, which dictates how much the error between the new and old estimate at each step should be taken into account. I use a value of 1.
- λ, the eligibility trace decay, which determines how much we update recently visited states given the new information we are learning online. I use a value of 0.8.
- ε, the exploration rate, i.e. at each step, the optimal action according to the current policy is chosen with probability $(1-\varepsilon)$, and a random action is chosen with probability ε. I use a value of 0 for now, in order to model a fully deterministic environment.

I find the optimal reward function using the algorithm suggested in the paper by Singh et al. (2009). [25] The implementation is described in Algorithm (1).

Algorithm 1 Optimal Reward Algorithm

```
for each element in the defined reward function space do
    sample 100 different environments randomly
    for each sampled environment do
        run Q-learning algorithm for 1 episode with 10,000 steps
        compute total fitness over episode history
    end for
    compute average fitness across environments
end for
```

Figure (2) shows the overall workflow of my implementation.

4.3. Major Challenges

I decided the size of the grid, the parameter values, the number of steps, number of environments to iterate over and the fitness function after playing around with a few options and checking what gave plausible results for simple cases. I assessed the plausibility of my results by running some base cases and plotting the learnt trajectory for each. These plots are shown in Figure (3). It makes a lot of sense that according to the learnt trajectory, the agent repeatedly steps in and out of the single food item which has an associated positive reward.

Several design choices I made were because what I had originally planned took too long to compute. Computational speed was the reason I had to change my design in the following ways:

- In setup 2, I was planning to iterate over the discrete range $[-5,6]$ for each of the 6 variables. Realizing this was taking too long, I tried the same range for 3 variables, but even that was too slow, i.e. the computation did not finish even after running for 48 hours straight. I then settled for 2 variables, and the discrete range $[-3,4]$.

- I ran the tests for a single fitness function, rather than collect data for multiple fitness functions.
- I ran tests for a single set of parameters, rather than collect data for multiple combinations of parameters.

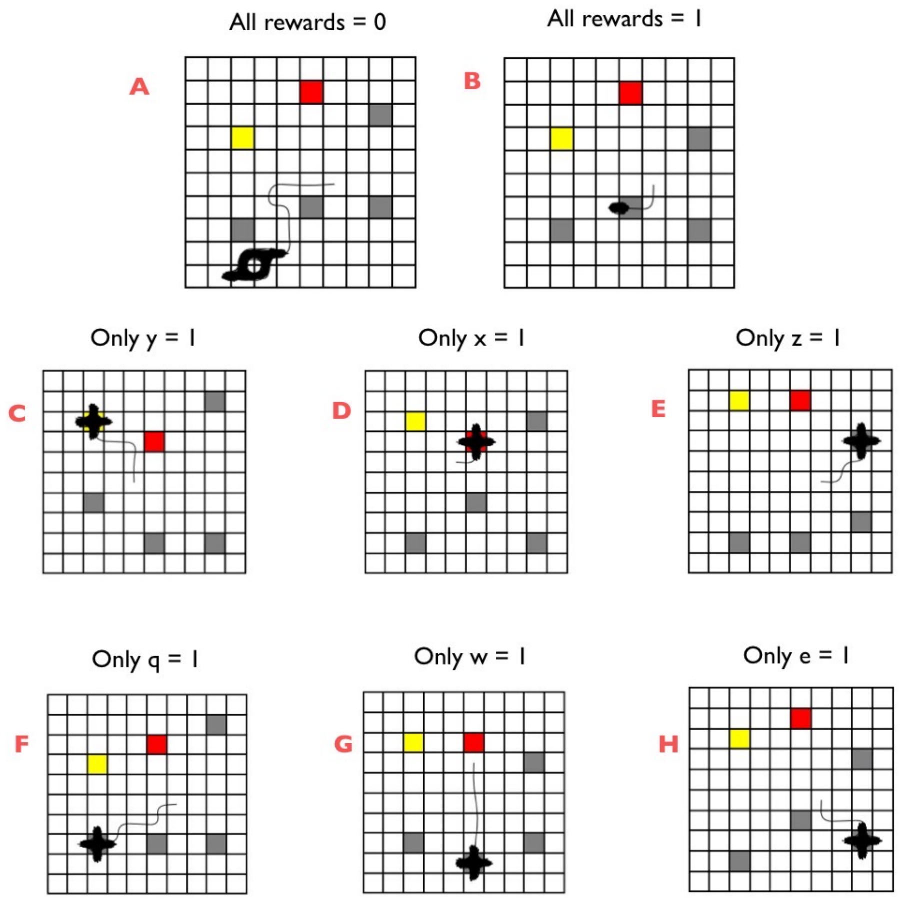


Figure 3: The diagram above shows the plotted trajectories for 7 different base cases. (**A**) All rewards are set to 0. (**B**) All rewards are set to 1. (**C**) Only y is set to 1, all other rewards are set to 0. (**D**) Only x is set to 1. (**E**) Only z is set to 1. (**F**) Only q is set to 1. (**G**) Only w is set to 1. (**H**) Only e is set to 1. In (**A**), the agent circles around a random 2 x 2 region. In (**B**), the agent gets to the closest food item and repeatedly steps in and out of it. In the rest of the 6 cases, the agent finds the unique food item with the reward set to 1, and then repeatedly steps in and out of the square until the end of its lifetime.

4.4. Evaluation

Table (1) shows a summary of the results from the 2 set ups. When computing the mean and variance for setup 1, I excluded the first case with all rewards set to zero, because this was the only negative value which unfairly skewed the statistics. Figure (4) shows a plot of fitness values of each tested reward function. In these plots its clear that the fitness value for zero rewards is drastically low (negative value), whereas the fitness values for all other reward functions, whether they have single or multiple rewards, is pretty similar. This fact is captured in the significantly low coefficient of variation of 0.165 for the fitness values across all reward functions (single and multiple). The maximum fitness for setup 1 was 3436.78, and this corresponded to the reward function where $x = 0, y = 0, z = 1, q = 1, w = 1, e = 0$. While this optimal reward function appears to have multiple rewards, the difference in the fitness value of this multi-reward function and the value of single-reward functions is very minimal. For example, the single-reward function where only y is set to 1, while the rest are set to 0 has an average fitness value of 3380.44, which is very close to 3436.78, the fitness of the optimal reward function. The full results are included in the appendix.

	Set up 1	**Set up 2**
Mean Fitness	3399.97	1962.87
Variance	559.96	3986013.35
Coefficient of Variance	0.165	2030.707
Maximum fitness	3436.78	3421.23
Optimal Reward-Function	[**x**:0, **y**:0, **z**:1, **q**:1, **w**:1, **e**:0]	[**y**:2, **e**:1]

Table 1: Summary of results for each set up. The mean and variance for set up 1 were computed excluding the first case where all rewards are set to 0, because that was the only large negative value which skewed the statistics. The mean, variance and coefficient of variance values for setup 2 appear uninformative because the data for setup 2 included either extremely high or extremely low values.

Set up 2 is slightly trickier to analyze. The summary of results gives us a maximum fitness of 3421.23 corresponding to an optimal reward function where $y = 2$ and $e = 1$. However, looking at the variance and coefficient of variation in this set up makes the results seem extremely unreliable. The mean and variance of the distribution of fitness values over the space of reward functions are

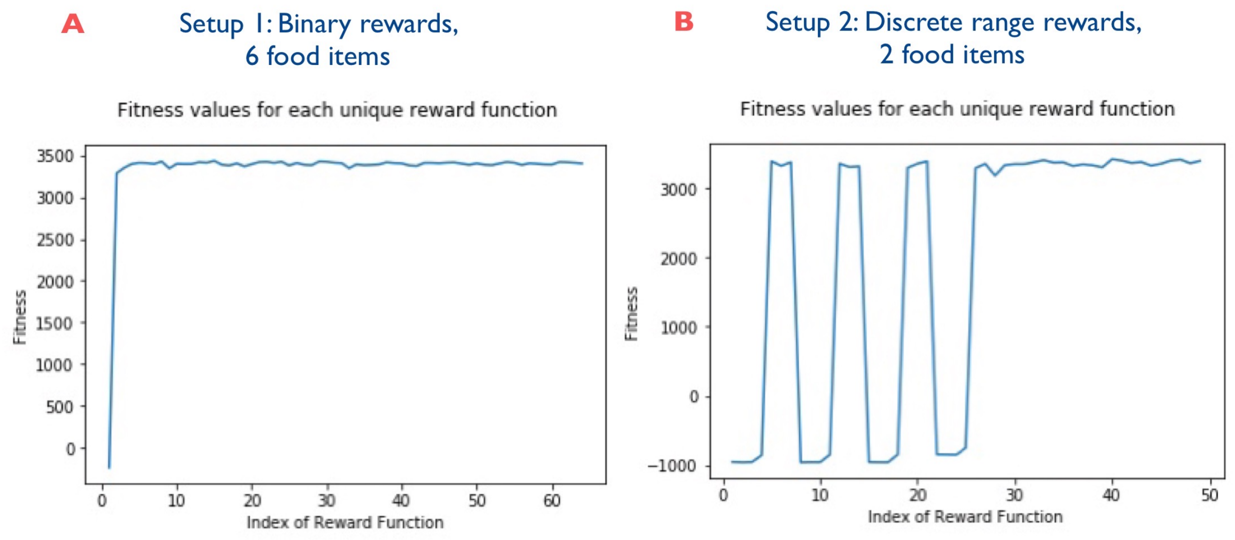


Figure 4: Plots of fitness values for each reward function that I tested. **(A)** shows the results for set up 1, where we had binary rewards and 6 food items. There were 64 possible reward functions. The first of these was when all food items had a reward value of 0. All the remaining functions were either single-reward or multi-reward. Notice in the plot that the only outlier value is the one for the very first function, with all rewards set to 0. In all other reward functions, the fitness values are high with little variation across single v. multiple rewards. **(B)** shows the results for set up 2. Every time both food items have a non-positive reward value, the fitness value is extremely negative. As soon as at least 1 of the items has a positive value, the fitness value spikes up to a high positive value. This plot appears to have a periodic pattern because of the algorithm I used to sample through the space of reward functions, as explained in Section 4.4.

uninformative metrics in this case. Instead, the plot in Figure (5) helps visualize the distribution of fitness values.

As long as both *y* and *e* are set to a non-positive value, the average fitness is significantly negative. As soon as either one of the two is set to a positive value, the average fitness is significantly high, similar to the average fitness achieved in set up 1. The plot in Figure (5) shows how the reward functions are separated into two sets, one where both rewards are non-positive, and the other where at least one reward is positive. Interestingly, *within* the two sets, the fitness values are arranged in a planar form, indicating that there is very little variation in fitness across reward functions that share the same set.

The plot in Figure (4 **B**) reveals a similar pattern in the fitness values. At first glance it appears that the fitness values seem to be periodically fluctuating between extremely high and extremely low values. However, the actual trend is simple, as also seen in Figure (5). The pattern seen in Figure (4 **B**) is periodic simply because of the algorithm I used to sample through all possible reward functions. With y fixed at -3, I iterate over all negative and positive values of e. When e is negative, fitness values are negative. When e is positive fitness values are positive. Once we have ran through all possible values of e, we change y to -2 and repeat. Thus, as long as y is set to a negative value, we see the periodic fluctuation in fitness values as e changes from negative to positive. Once y becomes positive, the fluctuation stops and all subsequent reward functions have positive fitness values.

These results make a lot of sense considering the findings from set up 1. While the maximum fitness is achieved by a multi-reward function (i.e. $y = 2$ and $e = 1$), the difference in fitness between this function and a single-reward function is minimal. The fitness when $y = 1$ and $e = -3$ is 3335.32 and when $y = 1$ and $e = 0$ is 3377.78. These values are very similar to the maximum fitness value of 3421.23.

Results from setup 2 reveal that having no reward functions, or negative reward functions are ineffective for maximizing fitness. However, as soon as we have a single positive reward function, introducing multiple rewards does not create a significant increase in fitness.

5. Summary

5.1. Conclusions

Results from both setups allow us to make two important conclusions.

(1) Rewards play a crucial role in maximizing fitness, as is evident by the negative fitness values whenever all rewards were set to 0 or, in the case of setup 2, a negative value.

(2) There is no significant increase in the fitness value between single-reward and multi-reward functions. In setup 1 this was evident by the similarity in values of all reward functions except the first where all rewards were set to 0. In setup 2 this was evident by the fact that rewards were negative as long as *both* the rewards were non-positive. As soon as at least one of the rewards was set to a positive value, the fitness value rose close to what it was in the optimal reward function i.e. [y:2, e:1]

These results, once again, indicate that single reward functions and multi reward functions do

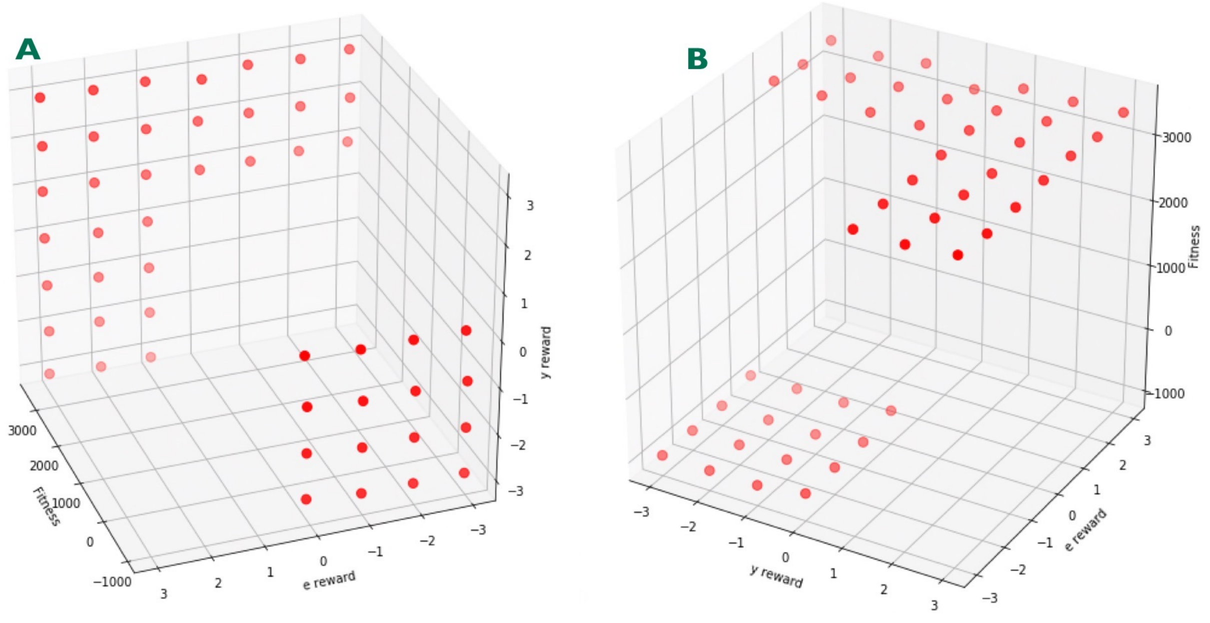


Figure 5: A 3-dimensional plot of the y and e values of a reward function against the corresponding fitness values. **(A)** and **(B)** are simply rotated versions of the same plot, just to help the reader fully visualize the pattern captured in the spread of fitness values. Notice that for the $y - e$ plane, the lower left quadrant, where both have non-positive values, corresponds to extremely negative fitness values ≈ -1000. The other 3 quadrants, where at least one of the 2 are set to a positive value, correspond to extremely high fitness values, ≈ 3500

not result in a significant difference in fitness values in such set ups. Thus we can conclude that multi-reward functions are unnecessary in fully deterministic environments.

5.2. Limitations and Future Work

The results reported here only apply to strictly deterministic environments. The main purpose of my research is to answer the puzzle: *"If our happiness and mental health is elevated in response to the reduction of emotional responses, i.e. desires, then why have humans developed so many desires in the first place?"*

Testing on deterministic environments is just the first step towards answering this puzzle. Next, I will conduct a similar analysis on

1. Action-stochastic environments, where the exploration rate, $\varepsilon \neq 0$
2. Reward-stochastic environments where the food items "deplete" and "replenish" over time
3. Multi-agent environments in both deterministic and stochastic settings

6. Acknowledgements

I sincerely thank Professor Tom Griffiths, for introducing me to the fascinating field of computational cognitive science in COS 306, Spring 2019, and for advising me through this project. My deepest respect for Rachit Dubey, who committed so much of his time and effort into helping me decide on a project, explaining the theoretical foundations of reinforcement learning, introducing me to the code base, helping me with the implementation, and genuinely getting me excited by the prospect of answering this puzzle. Thank you to Qasim Nadeem for his moral support and for helping me decide what figures to include in the write up. Thank you to Pulkit Singh for recommending the Computational Cognitive Science lab for my independent work. And of course, my deepest, profoundest, immeasurable gratitude for Dean Anne Caswell-Klein, to whom I owe all and any of my academic achievements at Princeton University.

References

[1] "Dissecting reinforcement learning." [Online]. Available: https://mpatacchiola.github.io/blog/2016/12/09/dissecting-reinforcement-learning.html

[2] "Historical views of suffering and societal betterment." [Online]. Available: http://worldsuffering.org/historical-views-of-suffering-and-societal-betterment/

[3] "The problem of desire." [Online]. Available: https://www.psychologytoday.com/us/blog/hide-and-seek/201411/the-problem-desire

[4] "Pyrlap (python reinforcement learning and planning library) on github." [Online]. Available: https://github.com/markkho/pyrlap

[5] "The science of motivation." [Online]. Available: https://www.apa.org/science/about/psa/2018/06/motivation

[6] D. Ackley and M. Littman, "Interactions between learning and evolution," *Artificial life II*, vol. 10, pp. 487–509, 1991.

[7] A. G. Barto, S. Singh, and N. Chentanez, "Intrinsically motivated learning of hierarchical collections of skills," in *Proceedings of the 3rd International Conference on Development and Learning*, 2004, pp. 112–19.

[8] R. A. Bressan and J. A. Crippa, "The role of dopamine in reward and pleasure behaviour–review of data from preclinical research," *Acta Psychiatrica Scandinavica*, vol. 111, pp. 14–21, 2005.

[9] J. E. Brommer, "The evolution of fitness in life-history theory," *Biological Reviews*, vol. 75, no. 3, pp. 377–404, 2000.

[10] N. Chentanez, A. G. Barto, and S. P. Singh, "Intrinsically motivated reinforcement learning," in *Advances in neural information processing systems*, 2005, pp. 1281–1288.

[11] K. A. Comtois *et al.*, "Effectiveness of dialectical behavior therapy in a community mental health center," *Cognitive and Behavioral Practice*, vol. 14, no. 4, pp. 406–414, 2007.

[12] E. L. Deci, R. Koestner, and R. M. Ryan, "A meta-analytic review of experiments examining the effects of extrinsic rewards on intrinsic motivation." *Psychological bulletin*, vol. 125, no. 6, p. 627, 1999.

[13] C. R. DeCou, K. A. Comtois, and S. J. Landes, "Dialectical behavior therapy is effective for the treatment of suicidal behavior: A meta-analysis," *Behavior therapy*, vol. 50, no. 1, pp. 60–72, 2019.

[14] T. R. Egnew, "Suffering, meaning, and healing: challenges of contemporary medicine," *The Annals of Family Medicine*, vol. 7, no. 2, pp. 170–175, 2009.

[15] R. A. Guzzo, "Types of rewards, cognitions, and work motivation," *Academy of Management Review*, vol. 4, no. 1, pp. 75–86, 1979.

[16] T. O. Haugen *et al.*, "The ideal free pike: 50 years of fitness-maximizing dispersal in windermere," *Proceedings of the Royal Society B: Biological Sciences*, vol. 273, no. 1604, pp. 2917–2924, 2006.

[17] C. H. Kahn, “Plato’s theory of desire,” *The review of metaphysics*, vol. 41, no. 1, pp. 77–103, 1987.

[18] S. Kliem, C. Kröger, and J. Kosfelder, “Dialectical behavior therapy for borderline personality disorder: a meta-analysis using mixed-effects modeling.” *Journal of consulting and clinical psychology*, vol. 78, no. 6, p. 936, 2010.

[19] I. B. Mauss *et al.*, “Can seeking happiness make people unhappy? paradoxical effects of valuing happiness.” *Emotion*, vol. 11, no. 4, p. 807, 2011.

[20] P.-Y. Oudeyer and F. Kaplan, “What is intrinsic motivation? a typology of computational approaches,” *Frontiers in neurorobotics*, vol. 1, p. 6, 2009.

[21] P. R. Pintrich, “A motivational science perspective on the role of student motivation in learning and teaching contexts.” *Journal of educational Psychology*, vol. 95, no. 4, p. 667, 2003.

[22] T. W. Robbins and B. J. Everitt, “Neurobehavioural mechanisms of reward and motivation,” *Current opinion in neurobiology*, vol. 6, no. 2, pp. 228–236, 1996.

[23] T. Savage, “Artificial motives: A review of motivation in artificial creatures,” *Connection Science*, vol. 12, no. 3-4, pp. 211–277, 2000.

[24] J. Schooler and C. A. Schreiber, “Experience, meta-consciousness, and the paradox of introspection,” *Journal of consciousness studies*, vol. 11, no. 7-8, pp. 17–39, 2004.

[25] S. Singh, R. L. Lewis, and A. G. Barto, “Where do rewards come from,” in *Proceedings of the annual conference of the cognitive science society*. Cognitive Science Society, 2009, pp. 2601–2606.

[26] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[27] E. Uchibe and K. Doya, “Finding intrinsic rewards by embodied evolution and constrained reinforcement learning,” *Neural Networks*, vol. 21, no. 10, pp. 1447–1455, 2008.

[28] C. F. Vorkapic, “Yoga and mental health: A dialogue between ancient wisdom and modern psychology,” *International journal of yoga*, vol. 9, no. 1, p. 67, 2016.

APPENDIX

Setup 1: Binary Rewards, 6 Food Items

May 17, 2020

```
[12]: import numpy
      import matplotlib
      import matplotlib.pyplot as plt
```

```
[2]: #Experiments to run: Exp 1 -- random initialization of agent and goal positions␣
      ↪for different environments and compute fitness
     #In static environment -- having more rewards may not necessarily be better
     #Exp 2 -- same experiment but now the different goals will disappear after␣
      ↪certain time-steps
     %matplotlib inline

     import matplotlib
     import time
     import random
     import numpy as np
     import pandas as pd
     import seaborn as sns

     import os
     import sys
     module_path = os.path.abspath(os.path.join('..'))
     if module_path not in sys.path:
         sys.path.append(module_path)

     from pyrlap.domains.gridworld import GridWorld
     from pyrlap.algorithms.qlearning import Qlearning
     from pyrlap.domains.gridworld.gridworldvis import visualize_trajectory
     import warnings
     warnings.filterwarnings('ignore')
```

```
[3]: # define the gridworld here

     def create_array():
         y_loc = (random.randint(1, 3), 2) #rewards only in 2nd half of grid world,␣
      ↪in different columns
         x_loc = (random.randint(1, 3), 5)
         z_loc = (random.randint(1, 3), 8)
```

```
    q_loc = (random.randint(6, 8), 2) #rewards only in 2nd half of grid world,␣
↪in different columns
    w_loc = (random.randint(6, 8), 5)
    e_loc = (random.randint(6, 8), 8)
    locs = [y_loc, x_loc, z_loc, q_loc, w_loc, e_loc]

    gw_array = ['..........', '..........', '..........', '..........', '........
↪..',
                '..........', '..........', '..........', '..........', '........
↪..',
                ]
    #n*n grid

    s = gw_array[x_loc[0]]
    new = list(s)
    new[x_loc[1]] = 'x'
    s = ''.join(new)
    gw_array[x_loc[0]] = s

    s = gw_array[y_loc[0]]
    new = list(s)
    new[y_loc[1]] = 'y'
    s = ''.join(new)
    gw_array[y_loc[0]] = s

    s = gw_array[z_loc[0]]
    new = list(s)
    new[z_loc[1]] = 'z'
    s = ''.join(new)
    gw_array[z_loc[0]] = s


    s = gw_array[q_loc[0]]
    new = list(s)
    new[q_loc[1]] = 'q'
    s = ''.join(new)
    gw_array[q_loc[0]] = s

    s = gw_array[w_loc[0]]
    new = list(s)
    new[w_loc[1]] = 'w'
    s = ''.join(new)
    gw_array[w_loc[0]] = s

    s = gw_array[e_loc[0]]
    new = list(s)
    new[e_loc[1]] = 'e'
    s = ''.join(new)
    gw_array[e_loc[0]] = s

    return gw_array, (locs)

def create_gridworld(rewards={'.':0, 'x':0, 'y':0, 'z':0, 'q':0, 'w':0, 'e':0}):
    gw_array, locs = create_array() #get the array and the location of goals to␣
 ↪compute fitness
    in_state = (random.randint(4, 6), random.randint(4, 6)) #initial location at␣
 ↪bottom left corner
    gw = GridWorld(gridworld_array=gw_array,
    init_state=in_state, #random initialization of the initial state
    feature_rewards=rewards) #the reward function
    return gw, locs

def learn(gw,m):
    np.random.seed(1234)
    all_run_data = []
    for i in range(1):
        params = {'learning_rate': 1,
              'eligibility_trace_decay': .8,
              'initial_qvalue': 10}
        qlearn = Qlearning(gw,
                       softmax_temp=.2,
                       discount_rate=.99,
                       **params)
        run_data = qlearn.train(episodes=1,
                            max_steps=m,
                            run_id=i,
                            return_run_data=True)
        for r in run_data:
            r.update(params)
        all_run_data.extend(run_data)

    return qlearn, (all_run_data)

def plot_learnt(qlearn, gw): #function to plot the learnt trajectory
    traj = qlearn.run(softmax_temp=0.0, randchoose=0.0, max_steps=1000)
    gwp = gw.plot()
    gwp.plot_trajectory(traj=[(s, a) for s, a, ns, r in traj])

def plot_history(gw, all_run_data):
    run_df = pd.DataFrame(all_run_data) #convert to pandas
    state = run_df["s"] #get states visited
    state_array = state.values #convert to matrix

    action = run_df["a"] #get actions taken
    action_array = action.values #convert to matrix

    gwp = gw.plot() #plot it
    gwp.plot_trajectory(traj=[(s, a) for s in state_array for a in action_array])

def compute_fitness(all_run_data, locs, gw): #function computes fitness over the␣
↪history (you can also compute fitness over the learnt policy)
    fitness = 0

    run_df = pd.DataFrame(all_run_data) #convert to pandas
    state = run_df["s"] #get states visited
    state_array = state.values #convert to matrix

    for i in range(len(state_array)):
        a = state_array[i]
        if((9-a[1],a[0]) in locs): #loop through the traj, if traj position is␣
↪not same as food position then fitness decreases, else increases by one
            fitness = fitness+1
        else:
            fitness = fitness-0.1
    return fitness
```

[8]:
```
## REWARDS = Binary, all combinations, THOUSAND steps, CENTER initial state
# Initial state = center
# Episodes = 50
# Trials = 10
# Steps = 1000
# Environments = 10

fitness_vals_thousand = np.zeros(64)
x_vals_thousand = np.zeros(64)
y_vals_thousand = np.zeros(64)
z_vals_thousand = np.zeros(64)
q_vals_thousand = np.zeros(64)
w_vals_thousand = np.zeros(64)
e_vals_thousand = np.zeros(64)

count = 0


for x in range(2):
    for y in range(2):
        for z in range(2):
                for q in range(2):
                    for w in range(2):
                        for e in range(2):
                                rewards={'.':0, 'x':x*10, 'y':y*10, 'z':z*10,␣
↪'q':q*10, 'w':w*10, 'e':e*10}
                                fit = []
                                for env in range(100): #Repeatedly sample lots␣
↪of environments
                                    gw, locs = create_gridworld(rewards) #create␣
↪gridworld here
                                    qlearn, all_run_data = learn(gw, 10000)   ␣
↪#Q-learn here
                                    fitness= compute_fitness(all_run_data, locs,␣
↪gw) #compute fitness here for the enviroment
                                    fit.append(fitness)
                                    if (env == 1):
                                        print("Rewards are set as:")
                                        print('x =', x, 'y =', y, 'z =', z, 'q␣
↪=', q, 'w =', w, 'e =', e)
                                        plot_learnt(qlearn, gw)


                                fitness_vals_thousand[count] = sum(fit)/
↪float(len(fit))
                                print("Average Fitness",␣
↪fitness_vals_thousand[count])
                                x_vals_thousand[count] = x
                                y_vals_thousand[count] = y
                                z_vals_thousand[count] = z
                                q_vals_thousand[count] = q
                                w_vals_thousand[count] = w
                                e_vals_thousand[count] = e
                                count += 1
```

```
Rewards are set as:
x = 0 y = 0 z = 0 q = 0 w = 0 e = 0
Average Fitness -239.66899999996826
Rewards are set as:
x = 0 y = 0 z = 0 q = 0 w = 0 e = 1
Average Fitness 3288.1960000003123
Rewards are set as:
x = 0 y = 0 z = 0 q = 0 w = 1 e = 0
Average Fitness 3352.32600000033
Rewards are set as:
x = 0 y = 0 z = 0 q = 0 w = 1 e = 1
Average Fitness 3398.372000000334
Rewards are set as:
x = 0 y = 0 z = 0 q = 1 w = 0 e = 0
Average Fitness 3413.1670000003323
```

```
Rewards are set as:
x = 0 y = 0 z = 0 q = 1 w = 0 e = 1
Average Fitness 3409.4490000003343
Rewards are set as:
x = 0 y = 0 z = 0 q = 1 w = 1 e = 0
Average Fitness 3400.055000000336
Rewards are set as:
x = 0 y = 0 z = 0 q = 1 w = 1 e = 1
Average Fitness 3428.4350000003365
Rewards are set as:
x = 0 y = 0 z = 1 q = 0 w = 0 e = 0
Average Fitness 3345.913000000319
Rewards are set as:
x = 0 y = 0 z = 1 q = 0 w = 0 e = 1
Average Fitness 3400.924000000327
Rewards are set as:
x = 0 y = 0 z = 1 q = 0 w = 1 e = 0
Average Fitness 3398.3280000003324
Rewards are set as:
x = 0 y = 0 z = 1 q = 0 w = 1 e = 1
Average Fitness 3398.911000000334
Rewards are set as:
x = 0 y = 0 z = 1 q = 1 w = 0 e = 0
Average Fitness 3420.9110000003316
Rewards are set as:
x = 0 y = 0 z = 1 q = 1 w = 0 e = 1
Average Fitness 3413.3540000003313
Rewards are set as:
x = 0 y = 0 z = 1 q = 1 w = 1 e = 0
Average Fitness 3436.784000000338
Rewards are set as:
x = 0 y = 0 z = 1 q = 1 w = 1 e = 1
Average Fitness 3389.990000000334
Rewards are set as:
x = 0 y = 1 z = 0 q = 0 w = 0 e = 0
Average Fitness 3380.4420000003283
Rewards are set as:
x = 0 y = 1 z = 0 q = 0 w = 0 e = 1
Average Fitness 3406.4020000003325
Rewards are set as:
x = 0 y = 1 z = 0 q = 0 w = 1 e = 0
Average Fitness 3369.8270000003313
Rewards are set as:
x = 0 y = 1 z = 0 q = 0 w = 1 e = 1
Average Fitness 3396.8980000003316
Rewards are set as:
x = 0 y = 1 z = 0 q = 1 w = 0 e = 0
Average Fitness 3422.5500000003335
```

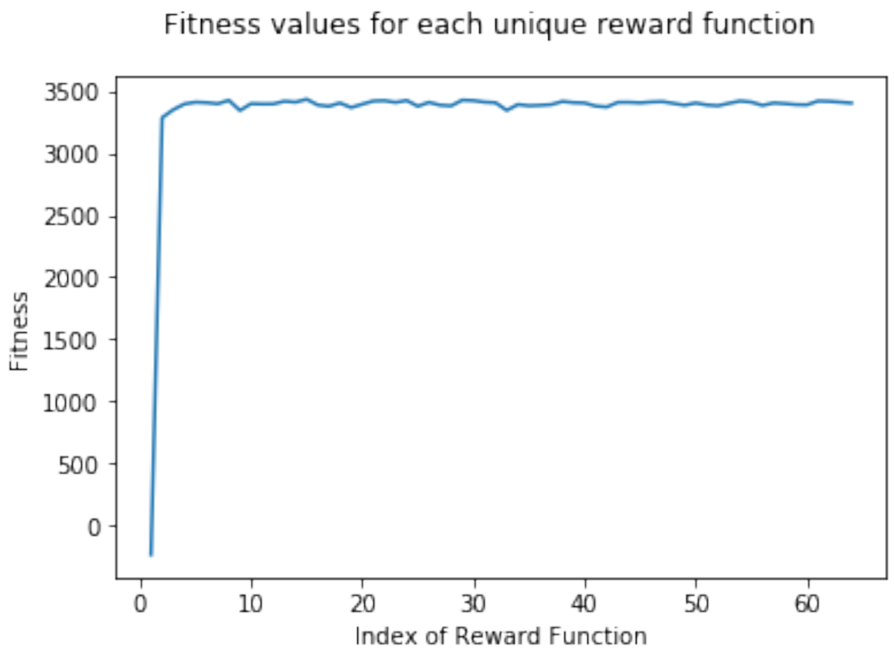
```
Rewards are set as:
x = 0 y = 1 z = 0 q = 1 w = 0 e = 1
Average Fitness 3425.278000000335
Rewards are set as:
x = 0 y = 1 z = 0 q = 1 w = 1 e = 0
Average Fitness 3411.4290000003357
Rewards are set as:
x = 0 y = 1 z = 0 q = 1 w = 1 e = 1
Average Fitness 3427.071000000335
Rewards are set as:
x = 0 y = 1 z = 1 q = 0 w = 0 e = 0
Average Fitness 3380.1670000003282
Rewards are set as:
x = 0 y = 1 z = 1 q = 0 w = 0 e = 1
Average Fitness 3411.9020000003325
Rewards are set as:
x = 0 y = 1 z = 1 q = 0 w = 1 e = 0
Average Fitness 3389.1430000003306
Rewards are set as:
x = 0 y = 1 z = 1 q = 0 w = 1 e = 1
Average Fitness 3383.665000000331
Rewards are set as:
x = 0 y = 1 z = 1 q = 1 w = 0 e = 0
Average Fitness 3429.898000000334
Rewards are set as:
x = 0 y = 1 z = 1 q = 1 w = 0 e = 1
Average Fitness 3425.8390000003355
Rewards are set as:
x = 0 y = 1 z = 1 q = 1 w = 1 e = 0
Average Fitness 3413.882000000334
Rewards are set as:
x = 0 y = 1 z = 1 q = 1 w = 1 e = 1
Average Fitness 3408.7340000003333
Rewards are set as:
x = 1 y = 0 z = 0 q = 0 w = 0 e = 0
Average Fitness 3346.859000000325
Rewards are set as:
x = 1 y = 0 z = 0 q = 0 w = 0 e = 1
Average Fitness 3394.31300000033
Rewards are set as:
x = 1 y = 0 z = 0 q = 0 w = 1 e = 0
Average Fitness 3384.600000000334
Rewards are set as:
x = 1 y = 0 z = 0 q = 0 w = 1 e = 1
Average Fitness 3387.9000000003325
Rewards are set as:
x = 1 y = 0 z = 0 q = 1 w = 0 e = 0
Average Fitness 3393.422000000333
```

```
Rewards are set as:
x = 1 y = 0 z = 0 q = 1 w = 0 e = 1
Average Fitness 3419.0520000003335
Rewards are set as:
x = 1 y = 0 z = 0 q = 1 w = 1 e = 0
Average Fitness 3409.801000000337
Rewards are set as:
x = 1 y = 0 z = 0 q = 1 w = 1 e = 1
Average Fitness 3406.468000000337
Rewards are set as:
x = 1 y = 0 z = 1 q = 0 w = 0 e = 0
Average Fitness 3381.740000000328
Rewards are set as:
x = 1 y = 0 z = 1 q = 0 w = 0 e = 1
Average Fitness 3374.5240000003264
Rewards are set as:
x = 1 y = 0 z = 1 q = 0 w = 1 e = 0
Average Fitness 3412.1330000003345
Rewards are set as:
x = 1 y = 0 z = 1 q = 0 w = 1 e = 1
Average Fitness 3412.2650000003355
Rewards are set as:
x = 1 y = 0 z = 1 q = 1 w = 0 e = 0
Average Fitness 3407.865000000333
Rewards are set as:
x = 1 y = 0 z = 1 q = 1 w = 0 e = 1
Average Fitness 3414.773000000332
Rewards are set as:
x = 1 y = 0 z = 1 q = 1 w = 1 e = 0
Average Fitness 3417.6110000003364
Rewards are set as:
x = 1 y = 0 z = 1 q = 1 w = 1 e = 1
Average Fitness 3403.3550000003365
Rewards are set as:
x = 1 y = 1 z = 0 q = 0 w = 0 e = 0
Average Fitness 3388.6590000003284
Rewards are set as:
x = 1 y = 1 z = 0 q = 0 w = 0 e = 1
Average Fitness 3406.43500000033
Rewards are set as:
x = 1 y = 1 z = 0 q = 0 w = 1 e = 0
Average Fitness 3390.5620000003332
Rewards are set as:
x = 1 y = 1 z = 0 q = 0 w = 1 e = 1
Average Fitness 3384.4790000003322
Rewards are set as:
x = 1 y = 1 z = 0 q = 1 w = 0 e = 0
Average Fitness 3404.3890000003325
```

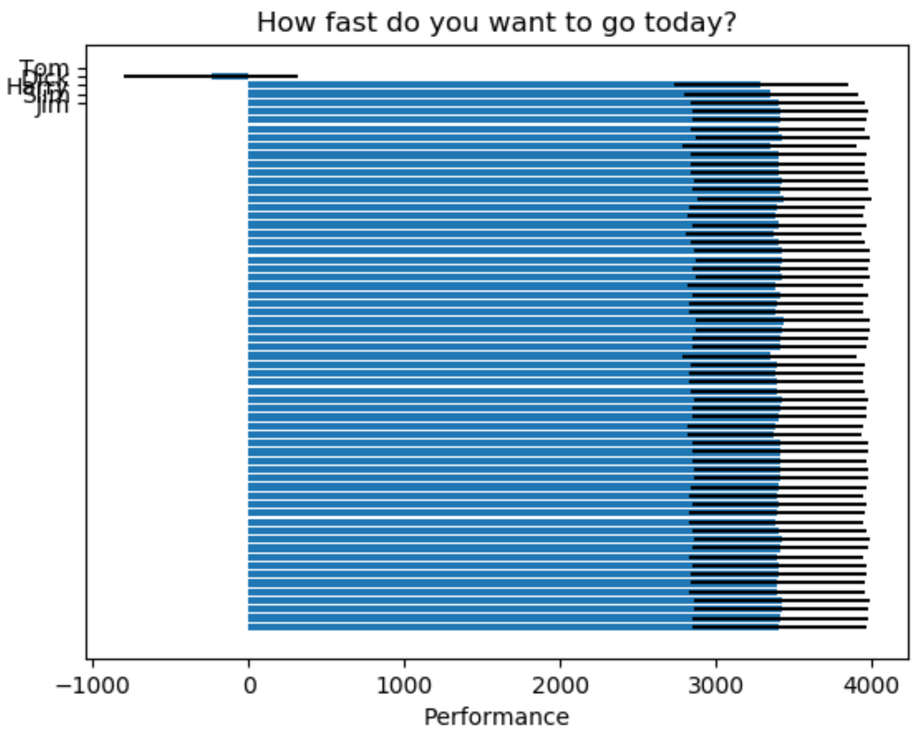
```
Rewards are set as:
x = 1 y = 1 z = 0 q = 1 w = 0 e = 1
Average Fitness 3422.605000000336
Rewards are set as:
x = 1 y = 1 z = 0 q = 1 w = 1 e = 0
Average Fitness 3413.717000000337
Rewards are set as:
x = 1 y = 1 z = 0 q = 1 w = 1 e = 1
Average Fitness 3387.0090000003333
Rewards are set as:
x = 1 y = 1 z = 1 q = 0 w = 0 e = 0
Average Fitness 3406.99600000033
Rewards are set as:
x = 1 y = 1 z = 1 q = 0 w = 0 e = 1
Average Fitness 3401.507000000334
Rewards are set as:
x = 1 y = 1 z = 1 q = 0 w = 1 e = 0
Average Fitness 3393.4660000003364
Rewards are set as:
x = 1 y = 1 z = 1 q = 0 w = 1 e = 1
Average Fitness 3391.7170000003307
Rewards are set as:
x = 1 y = 1 z = 1 q = 1 w = 0 e = 0
Average Fitness 3422.6270000003324
Rewards are set as:
x = 1 y = 1 z = 1 q = 1 w = 0 e = 1
Average Fitness 3419.7450000003364
Rewards are set as:
x = 1 y = 1 z = 1 q = 1 w = 1 e = 0
Average Fitness 3413.211000000337
Rewards are set as:
x = 1 y = 1 z = 1 q = 1 w = 1 e = 1
Average Fitness 3405.9400000003334
```

```
[19]: fig, axis = plt.subplots()

axis.plot(np.arange(1,65),fitness_vals_thousand)

fig.suptitle("Fitness values for each unique reward function", fontsize=12)
axis.set_xlabel('Index of Reward Function', fontsize=10)
axis.set_ylabel('Fitness', fontsize=10)

mean_center_init = np.mean(fitness_vals_thousand)
variance_center_init = np.var(fitness_vals_thousand)
cov_center_init = (variance_center_init/mean_center_init)*100

print("Mean Fitness over all reward functions = ", mean_center_init)
print("Variance of Fitness over all reward functions = ", variance_center_init)
print("Coefficient of Variation over all reward functions = ", cov_center_init)

mean_center_init = np.mean(fitness_vals_thousand[1:])
variance_center_init = np.var(fitness_vals_thousand[1:])
cov_center_init = (variance_center_init/mean_center_init)*100

print("Mean Fitness over all reward functions except case 1 with all rewards set␣
 ↪0 = ", mean_center_init)
print("Variance of Fitness over all reward functions except case 1 with all␣
 ↪rewards set 0 = ", variance_center_init)
print("Coefficient of variation over all reward functions except case 1 with all␣
 ↪rewards set 0 = ", cov_center_init)
```

```
Mean Fitness over all reward functions =  3343.0987187503274
Variance of Fitness over all reward functions =  204300.80709689224
Coefficient of Variation over all reward functions =  6111.120977404497
Mean Fitness over all reward functions except case 1 with all rewards set 0 =
3399.9680476193794
Variance of Fitness over all reward functions except case 1 with all rewards set
0 =  559.9608257597987
Coefficient of variation over all reward functions except case 1 with all
rewards set 0 =  16.469590829004325
```



[23]:
```
# Fixing random state for reproducibility
plt.rcdefaults()
fig, ax = plt.subplots()

# Example data
reward_functions = []
error = variance_center_init = np.var(fitness_vals_thousand[1:])


ax.barh(np.arange(1,65), fitness_vals_thousand, xerr=error, align='center')
ax.set_yticks(65)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_ylabel('Reward Function')
ax.set_xlabel('Fitness')

plt.show()
```



[24]:
```
max_fitness = np.max(fitness_vals_thousand)
max_index = np.where(fitness_vals_thousand == max_fitness)

max_x = x_vals_thousand[max_index[0][0]]
max_y = y_vals_thousand[max_index[0][0]]
max_z = z_vals_thousand[max_index[0][0]]
max_q = q_vals_thousand[max_index[0][0]]
max_w = w_vals_thousand[max_index[0][0]]
max_e = e_vals_thousand[max_index[0][0]]

#for i in range(len(max_index)):
    #max_y_e_tuples.append((y_vals_thousand[max_index[i]],␣
 ↪e_vals_thousand[max_index[i]]))


print(max_fitness)
print(type(max_index[0][0]))
print("Max x = ", max_x, "Max y = ", max_y, "Max z = ", max_z, "Max q = ",␣
 ↪max_q, "Max w = ", max_w, "Max e = ", max_e)
#print(max_y_e_tuples)
```

```
3436.784000000338
<class 'numpy.int64'>
Max x =  0.0 Max y =  0.0 Max z =  1.0 Max q =  1.0 Max w =  1.0 Max e =  0.0
```

```
[ ]:
```

Setup 2: Discrete range rewards, 2 food items

May 17, 2020

[54]:
```
#Experiments to run: Exp 1 -- random initialization of agent and goal positions␣
↪for different environments and compute fitness
#In static environment -- having more rewards may not necessarily be better
#Exp 2 -- same experiment but now the different goals will disappear after␣
↪certain time-steps
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt
import time
import random
import numpy as np
import pandas as pd
import seaborn as sns

import os
import sys
module_path = os.path.abspath(os.path.join('..'))
if module_path not in sys.path:
    sys.path.append(module_path)

from pyrlap.domains.gridworld import GridWorld
from pyrlap.algorithms.qlearning import Qlearning
from pyrlap.domains.gridworld.gridworldvis import visualize_trajectory
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
# define the gridworld here

def create_array():
    y_loc = (random.randint(1, 3), 2) #rewards only in 2nd half of grid world,␣
↪in different columns
    #x_loc = (random.randint(1, 3), 5)
    #z_loc = (random.randint(1, 3), 8)
    #q_loc = (random.randint(6, 8), 2) #rewards only in 2nd half of grid world,␣
↪in different columns
    #w_loc = (random.randint(6, 8), 5)
    e_loc = (random.randint(6, 8), 8)
    locs = [y_loc, e_loc]

    gw_array = ['..........', '..........', '..........', '..........', '........
↪..',
               '..........', '..........', '..........', '..........', '........
↪..',
               ]
    #n*n grid

    s = gw_array[y_loc[0]]
    new = list(s)
    new[y_loc[1]] = 'y'
    s = ''.join(new)
    gw_array[y_loc[0]] = s

    s = gw_array[e_loc[0]]
    new = list(s)
    new[e_loc[1]] = 'e'
    s = ''.join(new)
    gw_array[e_loc[0]] = s



    return gw_array, (locs)

'''

    s = gw_array[x_loc[0]]
    new = list(s)
    new[x_loc[1]] = 'x'
    s = ''.join(new)
    gw_array[x_loc[0]] = s

    s = gw_array[q_loc[0]]
    new = list(s)
    new[q_loc[1]] = 'q'
    s = ''.join(new)
    gw_array[q_loc[0]] = s

    s = gw_array[z_loc[0]]
    new = list(s)
    new[z_loc[1]] = 'z'
    s = ''.join(new)
    gw_array[z_loc[0]] = s
    s = gw_array[w_loc[0]]
    new = list(s)
    new[w_loc[1]] = 'w'
    s = ''.join(new)
    gw_array[w_loc[0]] = s


'''


def create_gridworld(rewards={'.':0, 'y':0, 'e':0}):
    gw_array, locs = create_array() #get the array and the location of goals to␣
 ↪compute fitness
    in_state = (random.randint(4, 6), random.randint(4, 6)) #initial location at␣
 ↪bottom left corner
    gw = GridWorld(gridworld_array=gw_array,
    init_state=in_state, #random initialization of the initial state
    feature_rewards=rewards) #the reward function
    return gw, locs

def learn(gw,m):
    np.random.seed(1234)
    all_run_data = []
    for i in range(1):
        params = {'learning_rate': 1,
              'eligibility_trace_decay': .8,
              'initial_qvalue': 10}
        qlearn = Qlearning(gw,
                       softmax_temp=.2,
                       discount_rate=.99,
                       **params)
        run_data = qlearn.train(episodes=1,
                            max_steps=m,
                            run_id=i,
                            return_run_data=True)
        for r in run_data:
            r.update(params)
        all_run_data.extend(run_data)

    return qlearn, (all_run_data)

def plot_learnt(qlearn, gw): #function to plot the learnt trajectory
    traj = qlearn.run(softmax_temp=0.0, randchoose=0.0, max_steps=1000)
    gwp = gw.plot()
    gwp.plot_trajectory(traj=[(s, a) for s, a, ns, r in traj])

def plot_history(gw, all_run_data):
    run_df = pd.DataFrame(all_run_data) #convert to pandas
    state = run_df["s"] #get states visited
    state_array = state.values #convert to matrix

    action = run_df["a"] #get actions taken
    action_array = action.values #convert to matrix

    gwp = gw.plot() #plot it
    gwp.plot_trajectory(traj=[(s, a) for s in state_array for a in action_array])

def compute_fitness(all_run_data, locs, gw): #function computes fitness over the␣
 ↪history (you can also compute fitness over the learnt policy)
    fitness = 0

    run_df = pd.DataFrame(all_run_data) #convert to pandas
    state = run_df["s"] #get states visited
    state_array = state.values #convert to matrix

    for i in range(len(state_array)):
        a = state_array[i]
        if((9-a[1],a[0]) in locs): #loop through the traj, if traj position is␣
 ↪not same as food position then fitness decreases, else increases by one
            fitness = fitness+1
        else:
            fitness = fitness-0.1
    return fitness
```

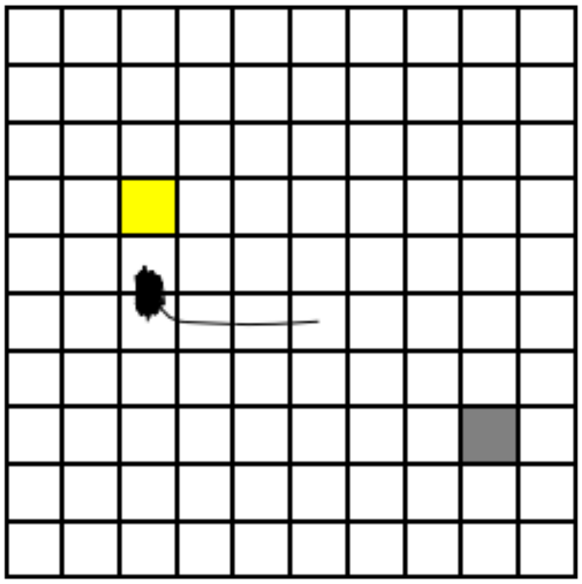
[4]:
```
## REWARDS = 3, Discrete range, all combinations, THOUSAND steps, CENTER initial␣
 ↪state
# Initial state = center
# Episodes = 50
# Trials = 10
# Steps = 1000
# Environments = 10

fitness_vals_thousand = []
y_vals_thousand = []
e_vals_thousand = []

for y in range(-3, 4):
    for e in range(-3, 4):
        rewards={'.':0, 'y':y*10, 'e':e*10}
        fit = []
        for env in range(10): #Repeatedly sample lots of environments
            gw, locs = create_gridworld(rewards) #create gridworld here
            qlearn, all_run_data = learn(gw, 10000)    #Q-learn here
            fitness= compute_fitness(all_run_data, locs, gw) #compute fitness␣
↪here for the enviroment
            fit.append(fitness)
            if (env == 1):
                print("Rewards are set as:")
                print('y =', y, 'e =', e)
                plot_learnt(qlearn, gw)


        fitness_vals_thousand.append(sum(fit)/float(len(fit)))
        y_vals_thousand.append(y)
        e_vals_thousand.append(e)
        print("Average Fitness", sum(fit)/float(len(fit)))
```

```
Rewards are set as:
y = -3 e = -3
Average Fitness -953.2500000001543
Rewards are set as:
y = -3 e = -2
Average Fitness -958.9700000001545
Rewards are set as:
y = -3 e = -1
Average Fitness -955.1200000001545
Rewards are set as:
y = -3 e = 0
Average Fitness -856.2300000001445
Rewards are set as:
y = -3 e = 1
Average Fitness 3390.760000000327
Rewards are set as:
y = -3 e = 2
Average Fitness 3324.540000000316
Rewards are set as:
y = -3 e = 3
Average Fitness 3376.5700000003294
Rewards are set as:
y = -2 e = -3
Average Fitness -959.9600000001541
Rewards are set as:
y = -2 e = -2
Average Fitness -956.4400000001548
Rewards are set as:
y = -2 e = -1
Average Fitness -957.4300000001547
Rewards are set as:
y = -2 e = 0
Average Fitness -848.530000000145
```

```
Rewards are set as:
y = -2 e = 1
Average Fitness 3357.4300000003254
Rewards are set as:
y = -2 e = 2
Average Fitness 3307.2700000003097
Rewards are set as:
y = -2 e = 3
Average Fitness 3316.9500000003195
Rewards are set as:
y = -1 e = -3
Average Fitness -955.6700000001543
Rewards are set as:
y = -1 e = -2
Average Fitness -958.200000000155
Rewards are set as:
y = -1 e = -1
Average Fitness -958.9700000001546
Rewards are set as:
y = -1 e = 0
Average Fitness -845.1200000001439
Rewards are set as:
y = -1 e = 1
Average Fitness 3295.7200000003186
Rewards are set as:
y = -1 e = 2
Average Fitness 3355.3400000003194
Rewards are set as:
y = -1 e = 3
Average Fitness 3390.430000000323
Rewards are set as:
y = 0 e = -3
Average Fitness -845.2300000001436
Rewards are set as:
y = 0 e = -2
Average Fitness -847.1000000001432
Rewards are set as:
y = 0 e = -1
Average Fitness -850.0700000001436
Rewards are set as:
y = 0 e = 0
Average Fitness -747.7700000001317
Rewards are set as:
y = 0 e = 1
Average Fitness 3292.2000000003122
Rewards are set as:
y = 0 e = 2
Average Fitness 3355.010000000326
```

```
Rewards are set as:
y = 0 e = 3
Average Fitness 3181.6500000003
Rewards are set as:
y = 1 e = -3
Average Fitness 3335.320000000325
Rewards are set as:
y = 1 e = -2
Average Fitness 3349.8400000003203
Rewards are set as:
y = 1 e = -1
Average Fitness 3349.730000000328
Rewards are set as:
y = 1 e = 0
Average Fitness 3377.7800000003285
Rewards are set as:
y = 1 e = 1
Average Fitness 3406.380000000335
Rewards are set as:
y = 1 e = 2
Average Fitness 3371.07000000033
Rewards are set as:
y = 1 e = 3
Average Fitness 3375.2500000003297
Rewards are set as:
y = 2 e = -3
Average Fitness 3322.450000000321
Rewards are set as:
y = 2 e = -2
Average Fitness 3346.760000000331
Rewards are set as:
y = 2 e = -1
Average Fitness 3333.2300000003256
Rewards are set as:
y = 2 e = 0
Average Fitness 3302.210000000324
Rewards are set as:
y = 2 e = 1
Average Fitness 3421.2300000003343
Rewards are set as:
y = 2 e = 2
Average Fitness 3400.2200000003286
Rewards are set as:
y = 2 e = 3
Average Fitness 3366.3400000003276
Rewards are set as:
y = 3 e = -3
Average Fitness 3379.760000000332
```

```
Rewards are set as:
y = 3 e = -2
Average Fitness 3327.6200000003228
Rewards are set as:
y = 3 e = -1
Average Fitness 3353.9100000003227
Rewards are set as:
y = 3 e = 0
Average Fitness 3399.0100000003317
Rewards are set as:
y = 3 e = 1
Average Fitness 3415.7300000003315
Rewards are set as:
y = 3 e = 2
Average Fitness 3362.160000000332
Rewards are set as:
y = 3 e = 3
Average Fitness 3394.8300000003283
```



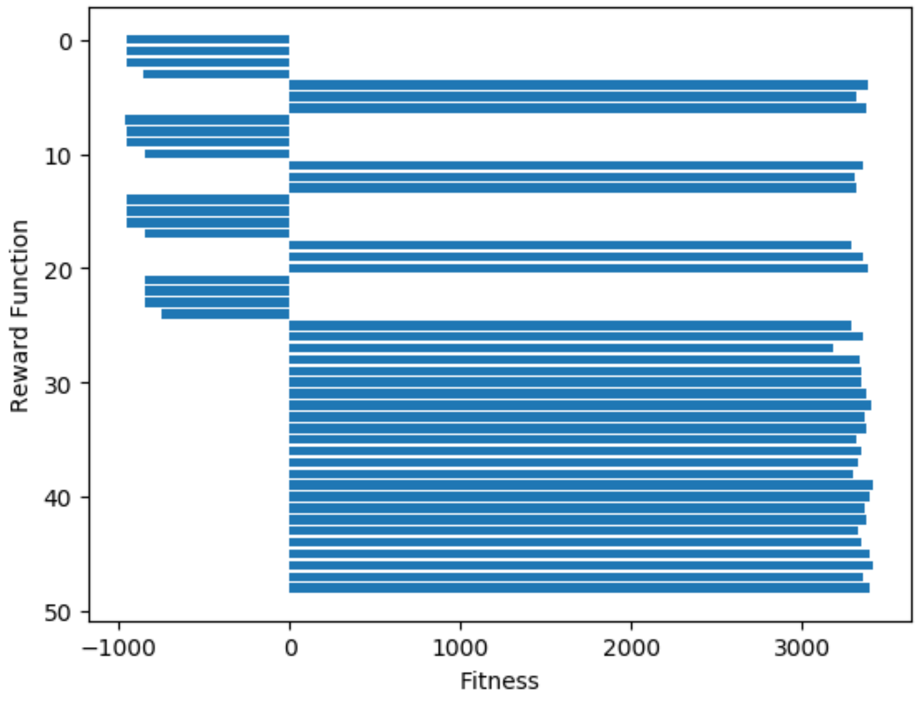
[ ]:

```
```

[16]:

```
plt.rcdefaults()
fig, ax = plt.subplots()

print(len(fitness_vals_thousand))
print(fitness_vals_thousand)
# Example data
reward_functions = []
error = np.var(fitness_vals_thousand[1:])


ax.barh(np.arange(0, len(fitness_vals_thousand)), fitness_vals_thousand,␣
 ↪align='center')
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_ylabel('Reward Function Index')
ax.set_xlabel('Fitness')

plt.show()
```

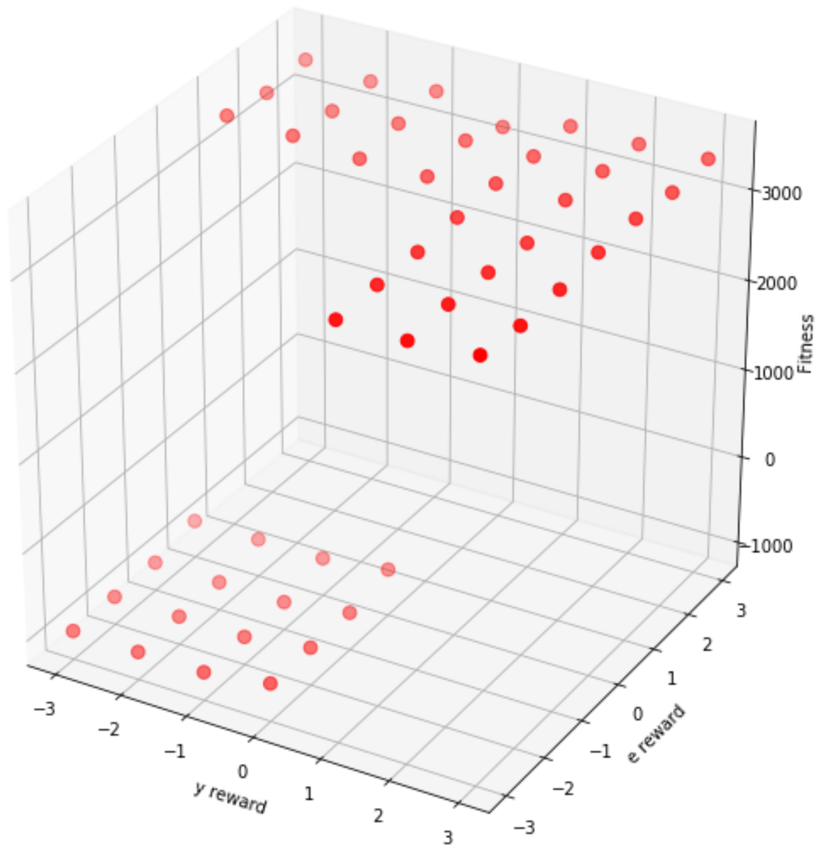
```
49
[-953.2500000001543, -958.9700000001545, -955.1200000001545, -856.2300000001445,
3390.760000000327, 3324.540000000316, 3376.5700000003294, -959.9600000001541,
-956.4400000001548, -957.4300000001547, -848.530000000145, 3357.4300000003254,
3307.2700000003097, 3316.9500000003195, -955.6700000001543, -958.200000000155,
-958.9700000001546, -845.1200000001439, 3295.7200000003186, 3355.3400000003194,
3390.430000000323, -845.2300000001436, -847.1000000001432, -850.0700000001436,
-747.7700000001317, 3292.2000000003122, 3355.010000000326, 3181.6500000003,
3335.320000000325, 3349.8400000003203, 3349.730000000328, 3377.7800000003285,
3406.380000000335, 3371.07000000033, 3375.2500000003297, 3322.450000000321,
3346.760000000331, 3333.2300000003256, 3302.210000000324, 3421.2300000003343,
3400.2200000003286, 3366.3400000003276, 3379.760000000332, 3327.6200000003228,
3353.9100000003227, 3399.0100000003317, 3415.7300000003315, 3362.160000000332,
3394.8300000003283]
```



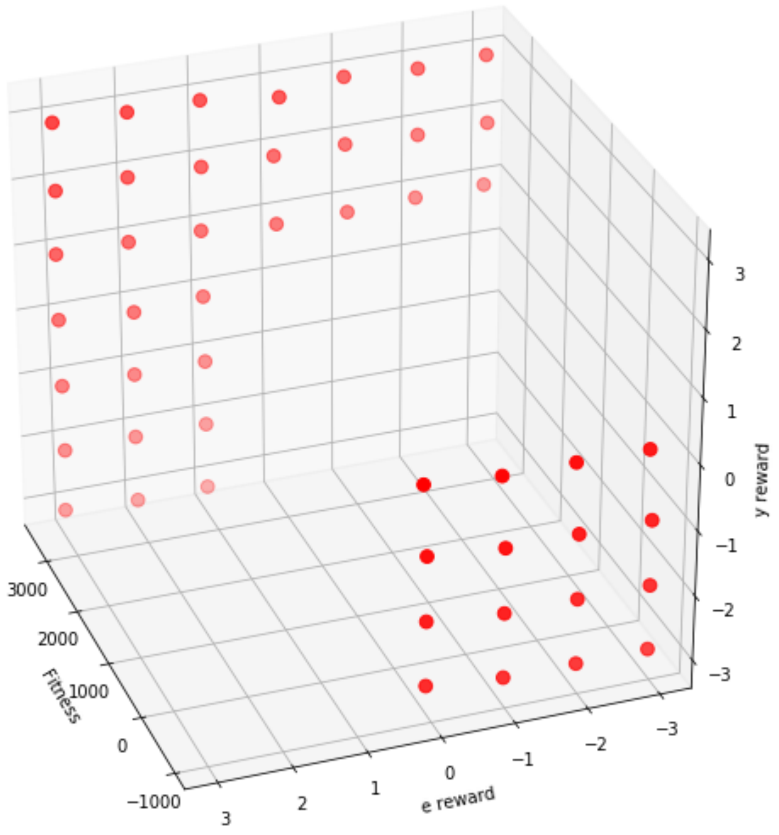
```
[32]: max_fitness = np.max(fitness_vals_thousand)
      max_index = np.where(fitness_vals_thousand == max_fitness)

      max_y = y_vals_thousand[max_index[0][0]]
      max_e = e_vals_thousand[max_index[0][0]]
      #for i in range(len(max_index)):
          #max_y_e_tuples.append((y_vals_thousand[max_index[i]],␣
       ↪e_vals_thousand[max_index[i]]))


      print(max_fitness)
      print(type(max_index[0][0]))
      print("Max y = ", max_y, "Max e = ", max_e)
      #print(max_y_e_tuples)
```

```
<class 'tuple'>
3421.2300000003343
<class 'numpy.int64'>
Max y =  2 Max e =  1
```

[80]:
```
fig = plt.figure(figsize=(10,10))
ax = plt.axes(projection="3d")

ax.scatter3D(y_vals_thousand, e_vals_thousand, fitness_vals_thousand, c='r',␣
 ↪marker='o', s=60)

ax.set_xlabel('y reward')
ax.set_ylabel('e reward')
ax.set_zlabel('Fitness')

plt.show()
```

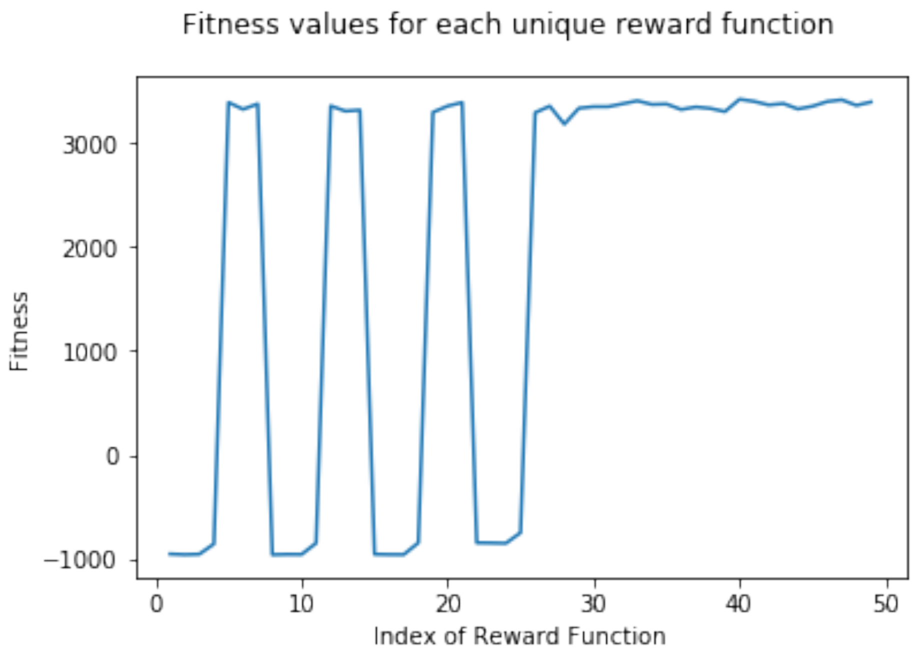


```
[79]: fig = plt.figure(figsize=(10,10))
      ax = plt.axes(projection="3d")

      ax.scatter3D(fitness_vals_thousand, y_vals_thousand, e_vals_thousand, c='r',␣
       ↪marker='o', s=60)

      ax.set_zlabel('y reward')
      ax.set_ylabel('e reward')
      ax.set_xlabel('Fitness')

      ax.view_init(30, 160)
      plt.draw()
```



[82]:
```
fig, axis = plt.subplots()

axis.plot(np.arange(1,len(fitness_vals_thousand)+1),fitness_vals_thousand)

fig.suptitle("Fitness values for each unique reward function", fontsize=12)
axis.set_xlabel('Index of Reward Function', fontsize=10)
axis.set_ylabel('Fitness', fontsize=10)

mean_center_init = np.mean(fitness_vals_thousand)
variance_center_init = np.var(fitness_vals_thousand)
cov_center_init = (variance_center_init/mean_center_init)*100

print("Mean Fitness over all reward functions = ", mean_center_init)
print("Variance of Fitness over all reward functions = ", variance_center_init)
print("Coefficient of Variation over all reward functions = ", cov_center_init)

mean_center_init = np.mean(fitness_vals_thousand[1:])
variance_center_init = np.var(fitness_vals_thousand[1:])
cov_center_init = (variance_center_init/mean_center_init)*100

print("Mean Fitness over all reward functions except case 1 with all rewards set␣
 ↪0 = ", mean_center_init)
print("Variance of Fitness over all reward functions except case 1 with all␣
 ↪rewards set 0 = ", variance_center_init)
print("Coefficient of variation over all reward functions except case 1 with all␣
 ↪rewards set 0 = ", cov_center_init)
```

```
Mean Fitness over all reward functions =  1962.8702040818025
Variance of Fitness over all reward functions =  3986013.3501988053
Coefficient of Variation over all reward functions =  203070.6534701002
Mean Fitness over all reward functions except case 1 with all rewards set 0 =
2023.62270833351
Variance of Fitness over all reward functions except case 1 with all rewards set
0 =  3888202.823124781
Coefficient of variation over all reward functions except case 1 with all
rewards set 0 =  192140.69930687753
```


Fitness values for each unique reward function
Fitness
Index of Reward Function
3000
2000
1000
0
−1000
0
10
20
30
40
50

```
[ ]:
```

Average Fitness Values in a Single-Reward v. Multi-Reward Environment

May 17, 2020

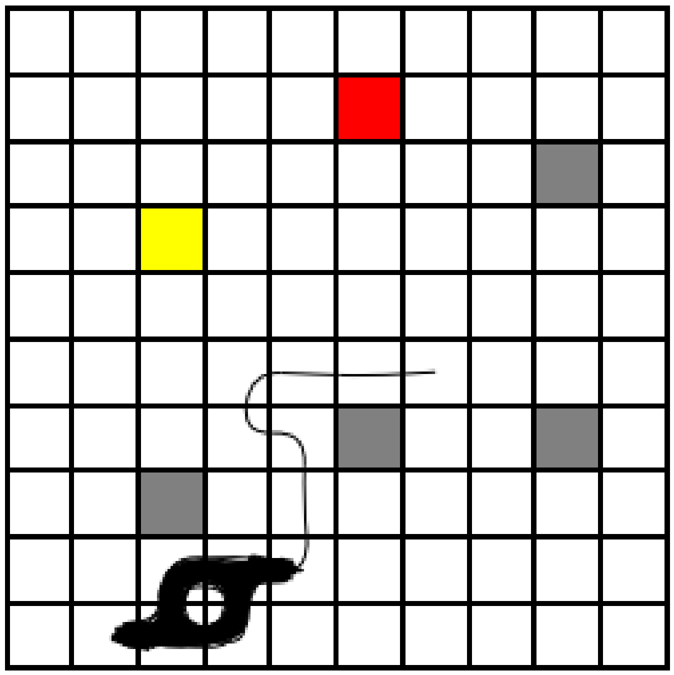
[19]:
```
#Experiments to run: Exp 1 -- random initialization of agent and goal positions␣
 ↪for different environments and compute fitness
#In static environment -- having more rewards may not necessarily be better
#Exp 2 -- same experiment but now the different goals will disappear after␣
 ↪certain time-steps
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt
import time
import random
import numpy as np
import pandas as pd
import seaborn as sns

import matplotlib.pyplot as plt

import os
import sys
module_path = os.path.abspath(os.path.join('..'))
if module_path not in sys.path:
    sys.path.append(module_path)

from pyrlap.domains.gridworld import GridWorld
from pyrlap.algorithms.qlearning import Qlearning
from pyrlap.domains.gridworld.gridworldvis import visualize_trajectory
import warnings
warnings.filterwarnings('ignore')
```

[20]:
```
# define the gridworld here

def create_array():
    y_loc = (random.randint(1, 3), 2) #rewards only in 2nd half of grid world,␣
 ↪in different columns
    x_loc = (random.randint(1, 3), 5)
    z_loc = (random.randint(1, 3), 8)
    q_loc = (random.randint(6, 8), 2) #rewards only in 2nd half of grid world,␣
↪in different columns
    w_loc = (random.randint(6, 8), 5)
    e_loc = (random.randint(6, 8), 8)
    locs = [y_loc, x_loc, z_loc, q_loc, w_loc, e_loc]

    gw_array = ['..........', '..........', '..........', '..........', '........
↪..',
                '..........', '..........', '..........', '..........', '........
↪..',
                ]
    #n*n grid

    s = gw_array[x_loc[0]]
    new = list(s)
    new[x_loc[1]] = 'x'
    s = ''.join(new)
    gw_array[x_loc[0]] = s

    s = gw_array[y_loc[0]]
    new = list(s)
    new[y_loc[1]] = 'y'
    s = ''.join(new)
    gw_array[y_loc[0]] = s

    s = gw_array[z_loc[0]]
    new = list(s)
    new[z_loc[1]] = 'z'
    s = ''.join(new)
    gw_array[z_loc[0]] = s


    s = gw_array[q_loc[0]]
    new = list(s)
    new[q_loc[1]] = 'q'
    s = ''.join(new)
    gw_array[q_loc[0]] = s

    s = gw_array[w_loc[0]]
    new = list(s)
    new[w_loc[1]] = 'w'
    s = ''.join(new)
    gw_array[w_loc[0]] = s

    s = gw_array[e_loc[0]]
    new = list(s)
    new[e_loc[1]] = 'e'
    s = ''.join(new)
    gw_array[e_loc[0]] = s

    return gw_array, (locs)

def create_gridworld(rewards={'.':0, 'x':0, 'y':0, 'z':0, 'q':10, 'w':0, 'e':0}):
    gw_array, locs = create_array() #get the array and the location of goals to␣
 ↪compute fitness
    in_state = (random.randint(4, 6), random.randint(4, 6)) #initial location at␣
 ↪bottom left corner
    gw = GridWorld(gridworld_array=gw_array,
    init_state=in_state, #random initialization of the initial state
    feature_rewards=rewards) #the reward function
    return gw, locs

def learn(gw,m):
    np.random.seed(1234)
    all_run_data = []
    for i in range(1):
        params = {'learning_rate': 1,
              'eligibility_trace_decay': .8,
              'initial_qvalue': 10}
        qlearn = Qlearning(gw,
                      softmax_temp=.2,
                      discount_rate=.99,
                      **params)
        run_data = qlearn.train(episodes=1,
                           max_steps=m,
                           run_id=i,
                           return_run_data=True)
        for r in run_data:
            r.update(params)
        all_run_data.extend(run_data)

    return qlearn, (all_run_data)

def plot_learnt(qlearn, gw): #function to plot the learnt trajectory
    traj = qlearn.run(softmax_temp=0.0, randchoose=0.0, max_steps=1000)
    gwp = gw.plot()
    gwp.plot_trajectory(traj=[(s, a) for s, a, ns, r in traj])

def compute_fitness(all_run_data, locs, gw): #function computes fitness over the␣
 ↪history (you can also compute fitness over the learnt policy)
    fitness = 0

    run_df = pd.DataFrame(all_run_data) #convert to pandas
    state = run_df["s"] #get states visited
    state_array = state.values #convert to matrix

    for i in range(len(state_array)):
        a = state_array[i]
        if((9-a[1],a[0]) in locs): #loop through the traj, if traj position is␣
↪not same as food position then fitness decreases, else increases by one
            fitness = fitness+1
        else:
            fitness = fitness-0.1
    return fitness
```

[21]:
```
fitness_vals = np.zeros(8)
x_labels = ['none', 'all', 'x', 'y', 'z', 'q', 'w', 'e']
```

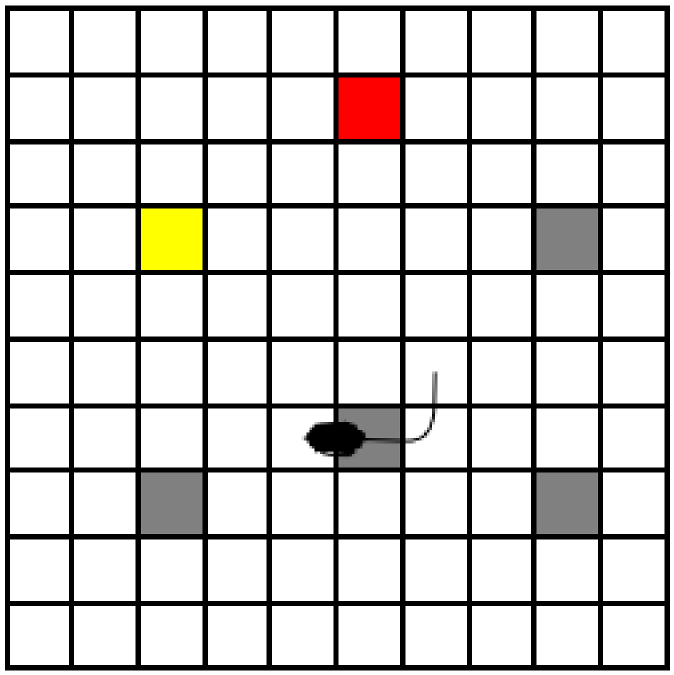
[22]:
```
## NO REWARDS ##

print("All 6 rewards = 0")

rewards={'.':0, 'x':0, 'y':0, 'z':0, 'q':0, 'w':0, 'e':0}

start_time = time.time()

gw, locs = create_gridworld(rewards)
qlearn, all_run_data = learn(gw, 10000)

fitness_vals[0] = compute_fitness(all_run_data, locs, gw)
print("Fitness = ", fitness_vals[0])

plot_learnt(qlearn, gw)

print("Execution Time", "--- %s seconds ---" % (time.time() - start_time))
```

```
All 6 rewards = 0
Fitness =  -207.999999999968
Execution Time --- 1.9911208152770996 seconds ---
```



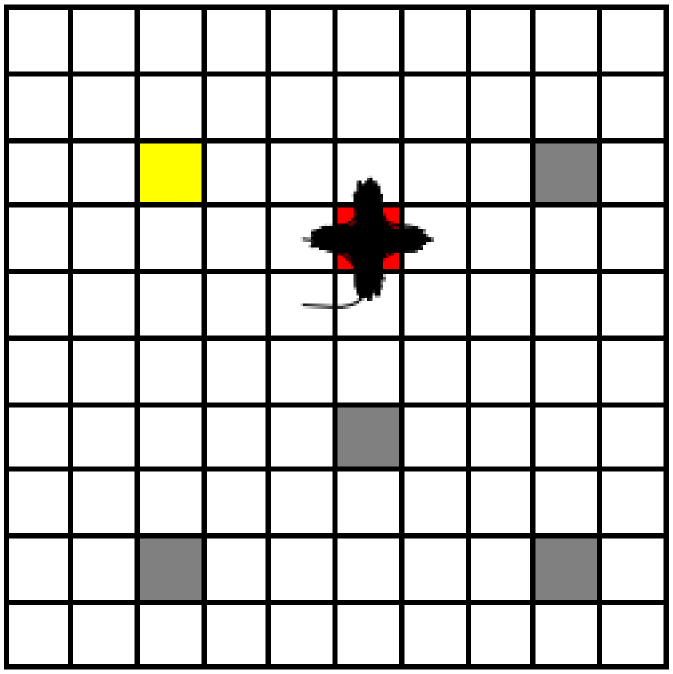
```
[23]: ## ALL REWARDS ##

      print("All 6 rewards = 1")

      rewards={'.':0, 'x':10, 'y':10, 'z':10, 'q':10, 'w':10, 'e':10}

      start_time = time.time()

      gw, locs = create_gridworld(rewards)
      qlearn, all_run_data = learn(gw, 10000)

      fitness_vals[1] = compute_fitness(all_run_data, locs, gw)
      print("Fitness = ", fitness_vals[1])

      plot_learnt(qlearn, gw)

      print("Execution Time", "--- %s seconds ---" % (time.time() - start_time))
```

```
All 6 rewards = 1
Fitness =  3486.900000000328
Execution Time --- 1.8676848411560059 seconds ---
```



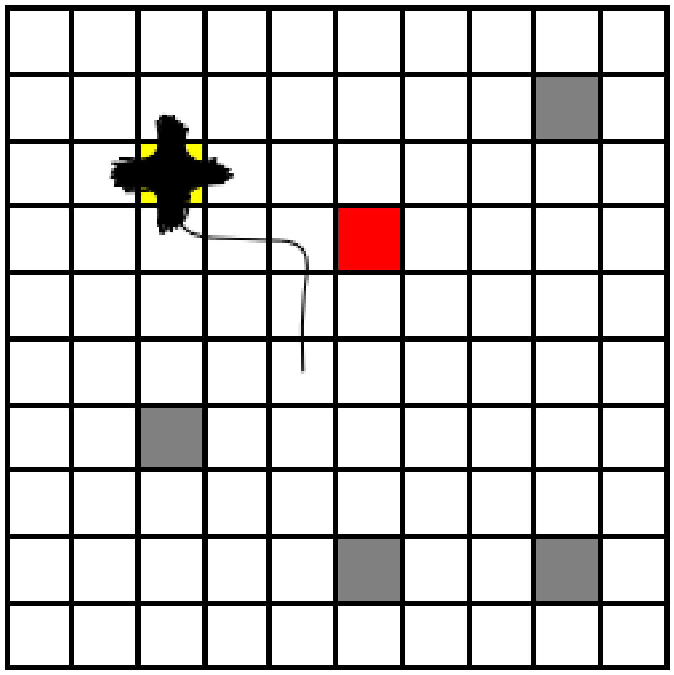
[33]:
```
## ONLY X = 1 ##

print("Only x = 1, all the rest 0")

rewards={'.':0, 'x':10, 'y':0, 'z':0, 'q':0, 'w':0, 'e':0}

start_time = time.time()

gw, locs = create_gridworld(rewards)
qlearn, all_run_data = learn(gw, 10000)

fitness_vals[2] = compute_fitness(all_run_data, locs, gw)
print("Fitness = ", fitness_vals[2])

plot_learnt(qlearn, gw)

print("Execution Time", "--- %s seconds ---" % (time.time() - start_time))
```

```
Only x = 1, all the rest 0
Fitness =  3345.0000000003324
Execution Time --- 2.0999321937561035 seconds ---
```



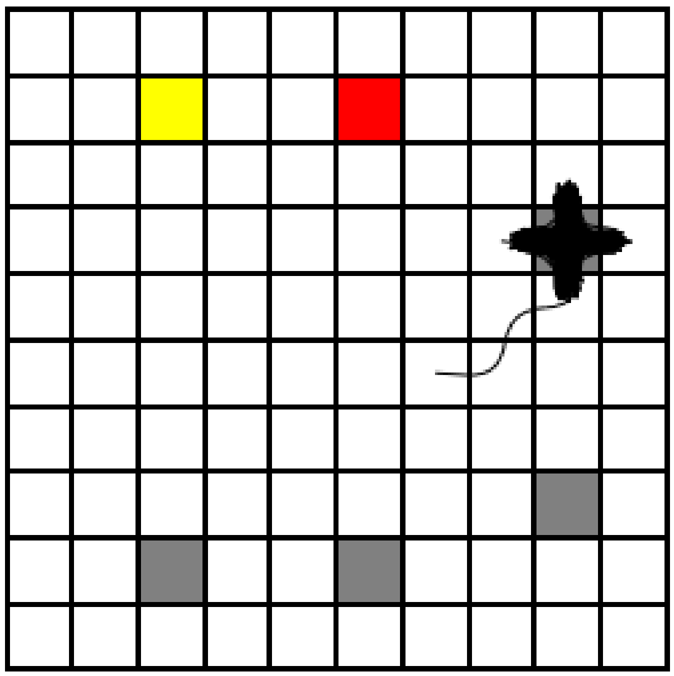
[34]:
```python
## ONLY Y = 1 ##

print("Only y = 1, all the rest 0")

rewards={'.':0, 'x':0, 'y':10, 'z':0, 'q':0, 'w':0, 'e':0}

start_time = time.time()

gw, locs = create_gridworld(rewards)
qlearn, all_run_data = learn(gw, 10000)

fitness_vals[3] = compute_fitness(all_run_data, locs, gw)
print("Fitness = ", fitness_vals[3])

plot_learnt(qlearn, gw)

print("Execution Time", "--- %s seconds ---" % (time.time() - start_time))
```

```
Only y = 1, all the rest 0
Fitness =  3457.2000000003336
Execution Time --- 1.9252688884735107 seconds ---
```



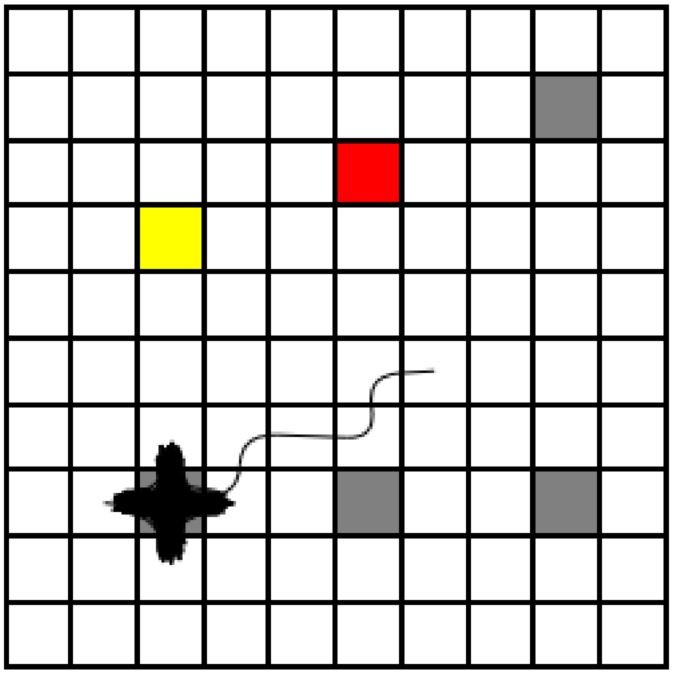
[36]:
```python
## ONLY Z = 1 ##

print("Only z = 1, all the rest 0")

rewards={'.':0, 'x':0, 'y':0, 'z':10, 'q':0, 'w':0, 'e':0}

start_time = time.time()

gw, locs = create_gridworld(rewards)
qlearn, all_run_data = learn(gw, 10000)

fitness_vals[4] = compute_fitness(all_run_data, locs, gw)
print("Fitness = ", fitness_vals[4])

plot_learnt(qlearn, gw)

print("Execution Time", "--- %s seconds ---" % (time.time() - start_time))
```

```
Only z = 1, all the rest 0
Fitness =  3150.300000000294
Execution Time --- 2.0427796840667725 seconds ---
```



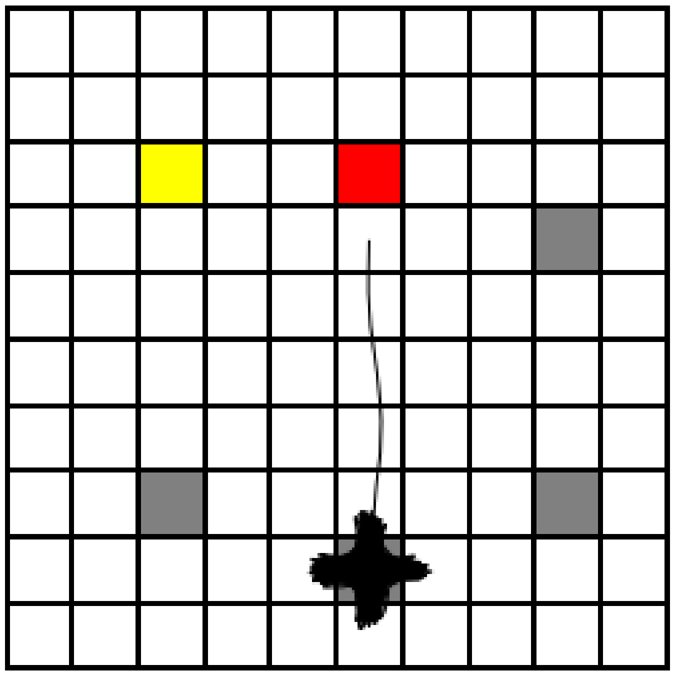
[37]:
```
## ONLY Q = 1 ##

print("Only q = 1, all the rest 0")

rewards={'.':0, 'x':0, 'y':0, 'z':0, 'q':10, 'w':0, 'e':0}

start_time = time.time()

gw, locs = create_gridworld(rewards)
qlearn, all_run_data = learn(gw, 10000)

fitness_vals[5] = compute_fitness(all_run_data, locs, gw)
print("Fitness = ", fitness_vals[5])

plot_learnt(qlearn, gw)

print("Execution Time", "--- %s seconds ---" % (time.time() - start_time))
```

```
Only q = 1, all the rest 0
Fitness =  3330.7000000003222
Execution Time --- 2.0776965618133545 seconds ---
```



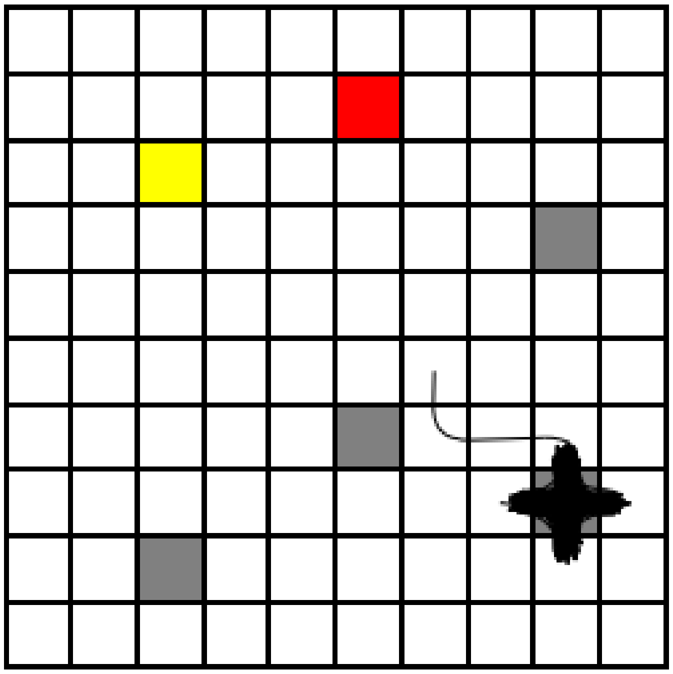
[28]:
```
## ONLY W = 1 ##

print("Only w = 1, all the rest 0")

rewards={'.':0, 'x':0, 'y':0, 'z':0, 'q':0, 'w':10, 'e':0}

start_time = time.time()

gw, locs = create_gridworld(rewards)
qlearn, all_run_data = learn(gw, 10000)

fitness_vals[6] = compute_fitness(all_run_data, locs, gw)
print("Fitness = ", fitness_vals[6])

plot_learnt(qlearn, gw)

print("Execution Time", "--- %s seconds ---" % (time.time() - start_time))
```

```
Only w = 1, all the rest 0
Fitness =  3386.800000000323
Execution Time --- 1.8348393440246582 seconds ---
```



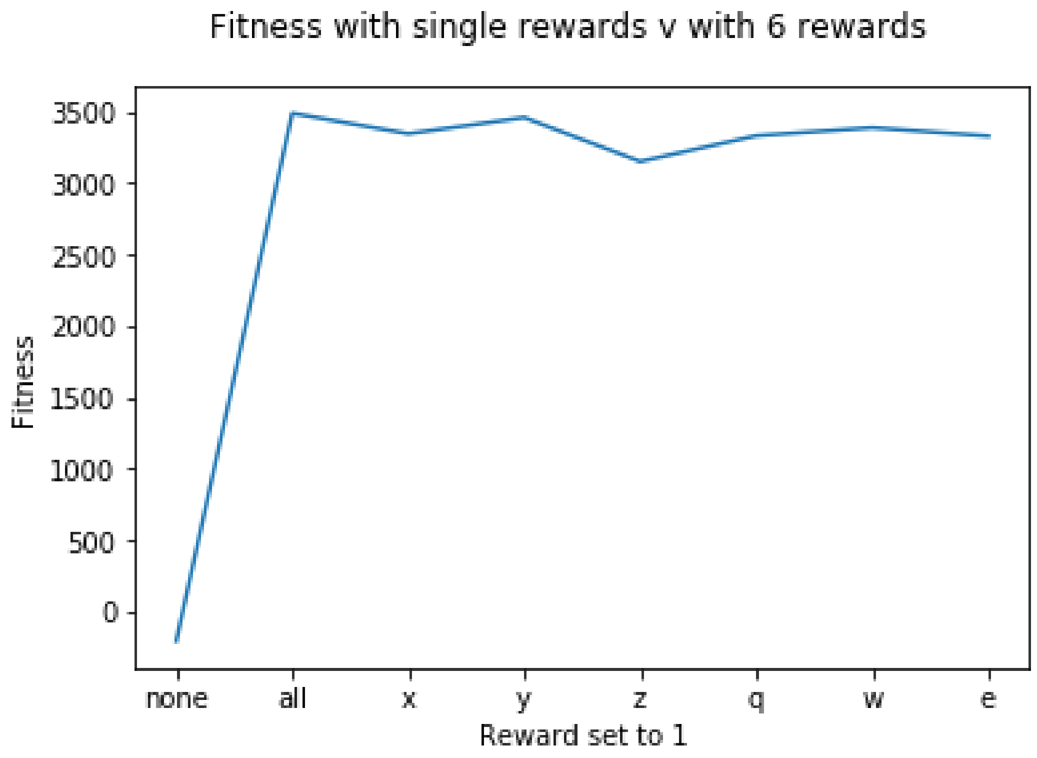
[38]:
```
## ONLY E = 1 ##

print("Only e = 1, all the rest 0")

rewards={'.':0, 'x':0, 'y':0, 'z':0, 'q':0, 'w':0, 'e':10}

start_time = time.time()

gw, locs = create_gridworld(rewards)
qlearn, all_run_data = learn(gw, 10000)

fitness_vals[7] = compute_fitness(all_run_data, locs, gw)
print("Fitness = ", fitness_vals[7])

plot_learnt(qlearn, gw)

print("Execution Time", "--- %s seconds ---" % (time.time() - start_time))
```

```
Only e = 1, all the rest 0
Fitness =  3328.5000000003256
Execution Time --- 2.083850622177124 seconds ---
```



```
[39]: fig, axis = plt.subplots()

      axis.plot(x_labels, fitness_vals)

      fig.suptitle("Fitness with single rewards v with 6 rewards", fontsize=12)
      axis.set_xlabel('Reward set to 1', fontsize=10)
      axis.set_ylabel('Fitness', fontsize=10)
```

```
[39]: Text(0, 0.5, 'Fitness')
```


Fitness with single rewards v with 6 rewards
Fitness
3500
3000
2500
2000
1500
1000
500
0
none
all
x
y
z
q
w
e
Reward set to 1

```
[ ]:
```

Miscellaneous: Binary Rewards, 2 Food Items

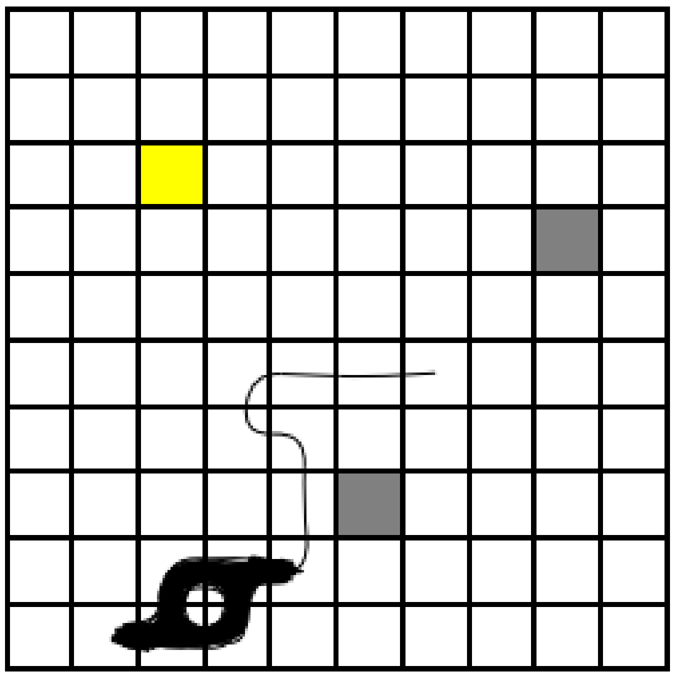
May 17, 2020

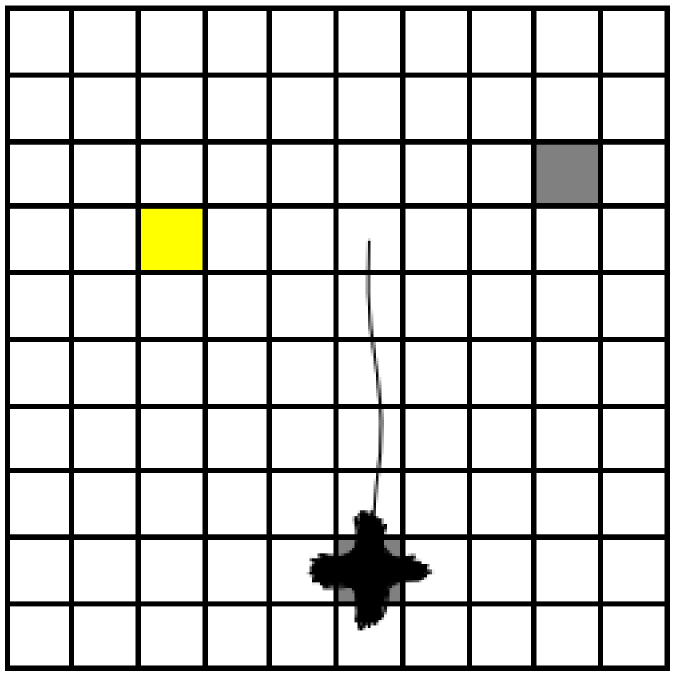
[9]:
```
#Experiments to run: Exp 1 -- random initialization of agent and goal positions␣
 ↪for different environments and compute fitness
#In static environment -- having more rewards may not necessarily be better
#Exp 2 -- same experiment but now the different goals will disappear after␣
 ↪certain time-steps
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt
import time
import random
import numpy as np
import pandas as pd
import seaborn as sns

import os
import sys
module_path = os.path.abspath(os.path.join('..'))
if module_path not in sys.path:
    sys.path.append(module_path)

from pyrlap.domains.gridworld import GridWorld
from pyrlap.algorithms.qlearning import Qlearning
from pyrlap.domains.gridworld.gridworldvis import visualize_trajectory
import warnings
warnings.filterwarnings('ignore')
```

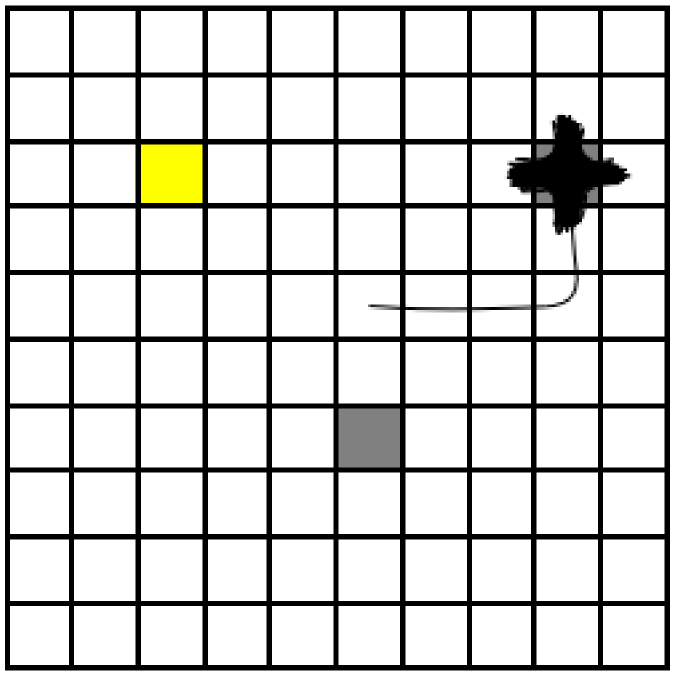
[2]:
```
# define the gridworld here

def create_array():
    y_loc = (random.randint(1, 3), 2) #rewards only in 2nd half of grid world,␣
 ↪in different columns
    #x_loc = (random.randint(1, 3), 5)
    z_loc = (random.randint(1, 3), 8)
    #q_loc = (random.randint(6, 8), 2) #rewards only in 2nd half of grid world,␣
 ↪in different columns
    w_loc = (random.randint(6, 8), 5)
    #e_loc = (random.randint(6, 8), 8)
    locs = [y_loc, z_loc, w_loc]

    gw_array = ['..........', '..........', '..........', '..........', '........
↪..',
                '..........', '..........', '..........', '..........', '........
↪..',
               ]
    #n*n grid

    s = gw_array[y_loc[0]]
    new = list(s)
    new[y_loc[1]] = 'y'
    s = ''.join(new)
    gw_array[y_loc[0]] = s

    s = gw_array[z_loc[0]]
    new = list(s)
    new[z_loc[1]] = 'z'
    s = ''.join(new)
    gw_array[z_loc[0]] = s


    s = gw_array[w_loc[0]]
    new = list(s)
    new[w_loc[1]] = 'w'
    s = ''.join(new)
    gw_array[w_loc[0]] = s

    return gw_array, (locs)

'''

    s = gw_array[x_loc[0]]
    new = list(s)
    new[x_loc[1]] = 'x'
    s = ''.join(new)
    gw_array[x_loc[0]] = s

    s = gw_array[q_loc[0]]
    new = list(s)
    new[q_loc[1]] = 'q'
    s = ''.join(new)
    gw_array[q_loc[0]] = s


    s = gw_array[e_loc[0]]
    new = list(s)
    new[e_loc[1]] = 'e'
    s = ''.join(new)
    gw_array[e_loc[0]] = s

'''


def create_gridworld(rewards={'.':0, 'y':0, 'z':0, 'w':0}):
    gw_array, locs = create_array() #get the array and the location of goals to␣
 ↪compute fitness
    in_state = (random.randint(4, 6), random.randint(4, 6)) #initial location at␣
 ↪bottom left corner
    gw = GridWorld(gridworld_array=gw_array,
    init_state=in_state, #random initialization of the initial state
    feature_rewards=rewards) #the reward function
    return gw, locs

def learn(gw,m):
    np.random.seed(1234)
    all_run_data = []
    for i in range(1):
        params = {'learning_rate': 1,
              'eligibility_trace_decay': .8,
              'initial_qvalue': 10}
        qlearn = Qlearning(gw,
                       softmax_temp=.2,
                       discount_rate=.99,
                       **params)
        run_data = qlearn.train(episodes=1,
                            max_steps=m,
                            run_id=i,
                            return_run_data=True)
        for r in run_data:
            r.update(params)
        all_run_data.extend(run_data)

    return qlearn, (all_run_data)

def plot_learnt(qlearn, gw): #function to plot the learnt trajectory
    traj = qlearn.run(softmax_temp=0.0, randchoose=0.0, max_steps=1000)
    gwp = gw.plot()
    gwp.plot_trajectory(traj=[(s, a) for s, a, ns, r in traj])

def plot_history(gw, all_run_data):
    run_df = pd.DataFrame(all_run_data) #convert to pandas
    state = run_df["s"] #get states visited
    state_array = state.values #convert to matrix

    action = run_df["a"] #get actions taken
    action_array = action.values #convert to matrix

    gwp = gw.plot() #plot it
    gwp.plot_trajectory(traj=[(s, a) for s in state_array for a in action_array])

def compute_fitness(all_run_data, locs, gw): #function computes fitness over the␣
 ↪history (you can also compute fitness over the learnt policy)
    fitness = 0

    run_df = pd.DataFrame(all_run_data) #convert to pandas
    state = run_df["s"] #get states visited
    state_array = state.values #convert to matrix

    for i in range(len(state_array)):
        a = state_array[i]
        if((9-a[1],a[0]) in locs): #loop through the traj, if traj position is␣
 ↪not same as food position then fitness decreases, else increases by one
            fitness = fitness+1
        else:
            fitness = fitness-0.1
    return fitness
```

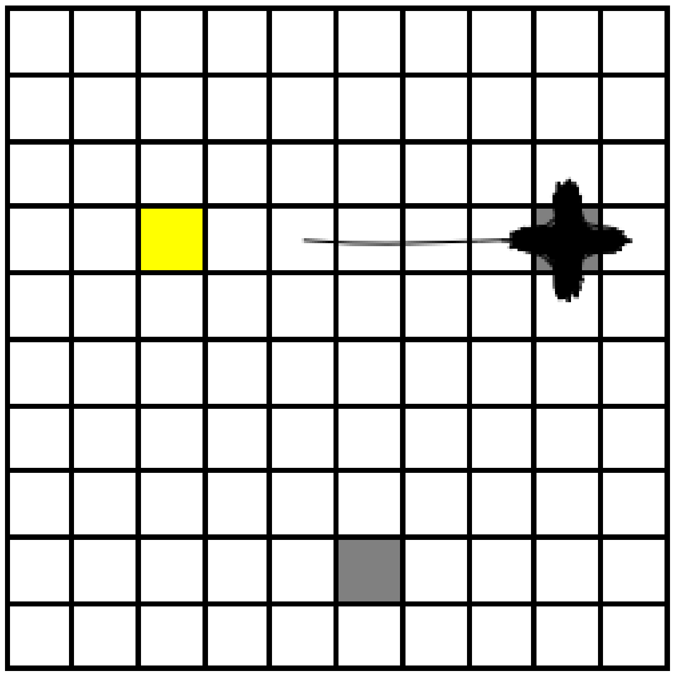
[3]:
```
## REWARDS = 3, Discrete range, all combinations, THOUSAND steps, CENTER initial␣
 ↪state
# Initial state = center
# Episodes = 50
# Trials = 10
# Steps = 1000
# Environments = 10

fitness_vals_thousand = []
y_vals_thousand = []
z_vals_thousand = []
w_vals_thousand = []

for y in range(2):
    for z in range(2):
        for w in range(2):
            rewards={'.':0, 'y':y*10, 'z':z*10, 'w':w*10}
            fit = []
            for env in range(100): #Repeatedly sample lots of environments
                gw, locs = create_gridworld(rewards) #create gridworld here
                qlearn, all_run_data = learn(gw, 10000)    #Q-learn here
                fitness= compute_fitness(all_run_data, locs, gw) #compute␣
↪fitness here for the enviroment
                fit.append(fitness)
                if (env == 1):
                    print("Rewards are set as:")
                    print('y =', y, 'z =', z,'w =', w)
                    plot_learnt(qlearn, gw)


            fitness_vals_thousand.append(sum(fit)/float(len(fit)))
            y_vals_thousand.append(y)
            z_vals_thousand.append(z)
            w_vals_thousand.append(w)
            print("Average Fitness", sum(fit)/float(len(fit)))
```

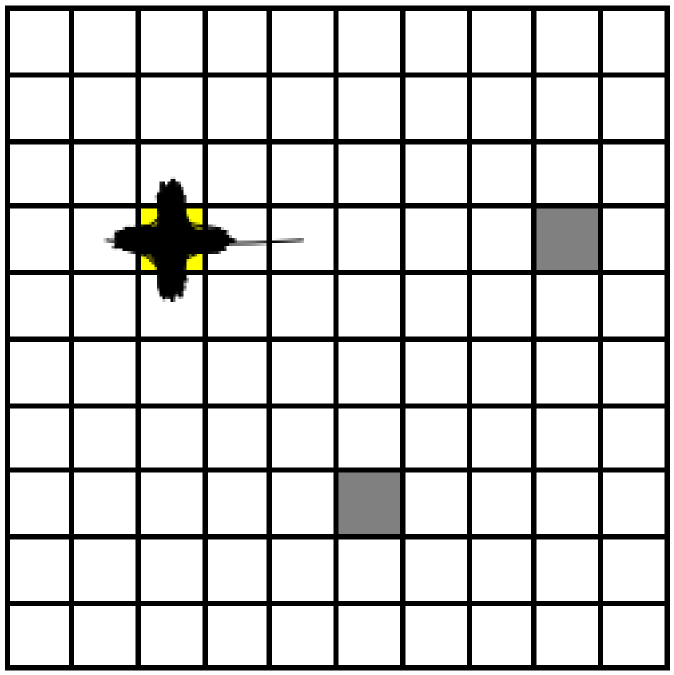
```
Rewards are set as:
y = 0 z = 0 w = 0
Average Fitness -621.2700000001139
Rewards are set as:
y = 0 z = 0 w = 1
Average Fitness 3322.0650000003266
Rewards are set as:
y = 0 z = 1 w = 0
Average Fitness 3311.692000000314
Rewards are set as:
y = 0 z = 1 w = 1
Average Fitness 3364.184000000328
Rewards are set as:
y = 1 z = 0 w = 0
Average Fitness 3366.3180000003254
Rewards are set as:
y = 1 z = 0 w = 1
Average Fitness 3359.773000000331
Rewards are set as:
y = 1 z = 1 w = 0
Average Fitness 3371.2900000003265
Rewards are set as:
y = 1 z = 1 w = 1
Average Fitness 3380.0680000003313
```

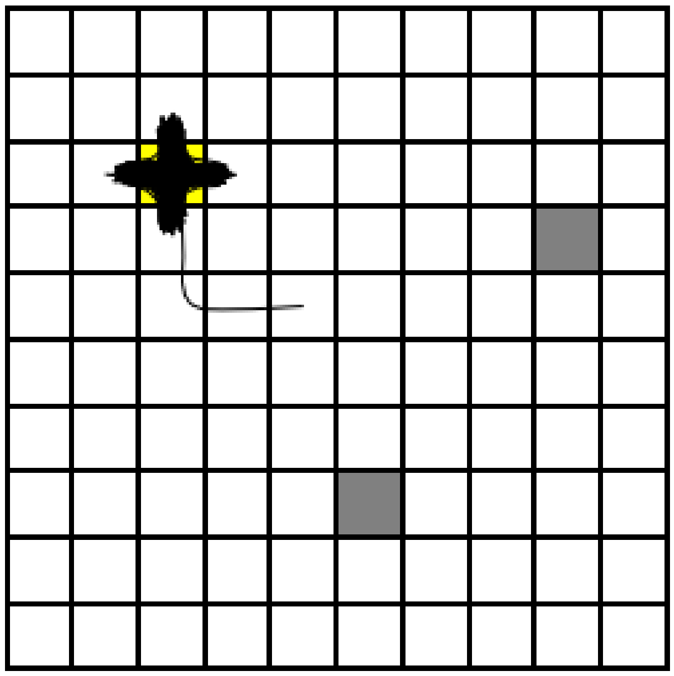


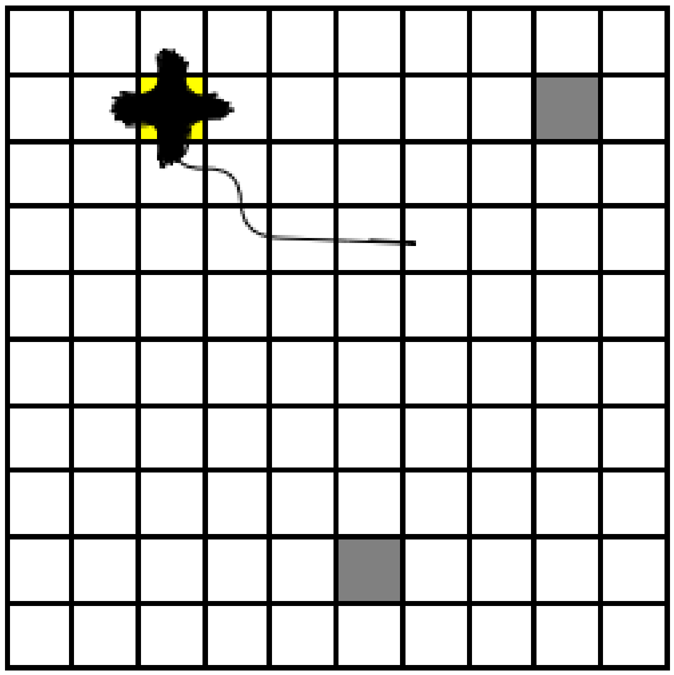


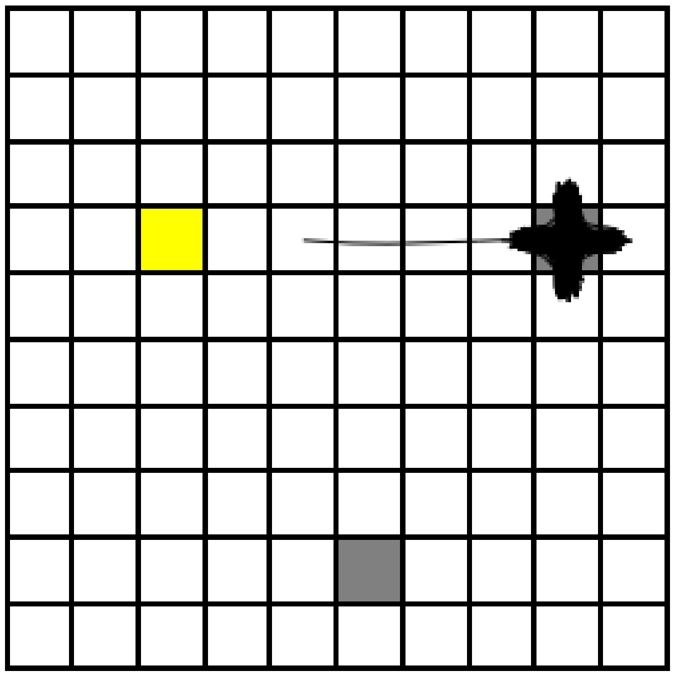










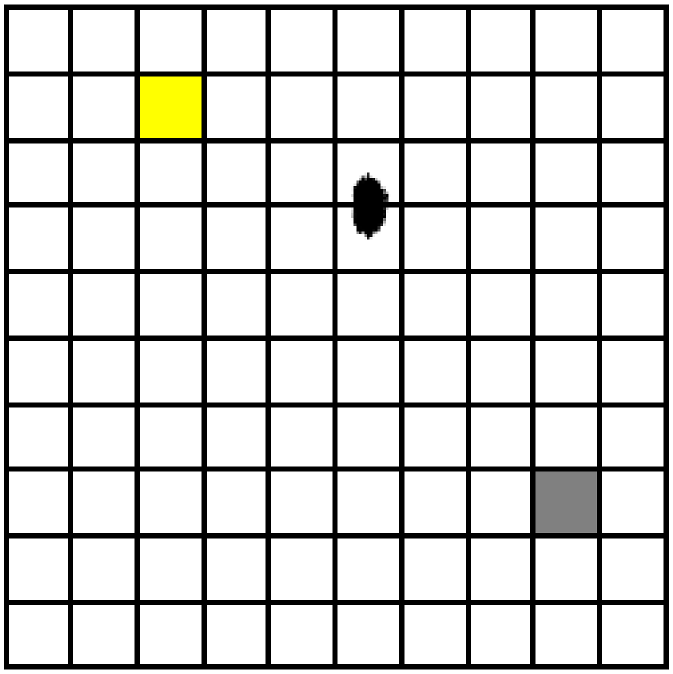


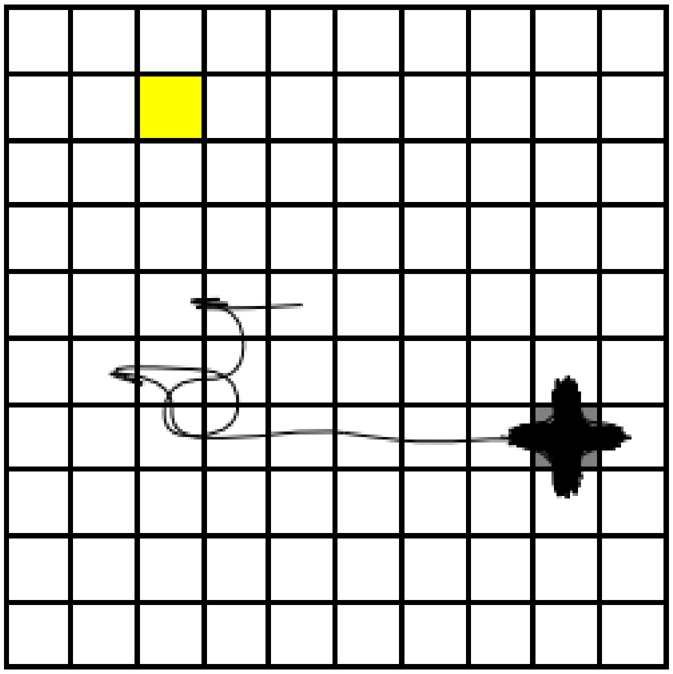
```
[4]: # define the gridworld here

def create_array():
    y_loc = (random.randint(1, 3), 2) #rewards only in 2nd half of grid world,␣
 ↪in different columns
    #x_loc = (random.randint(1, 3), 5)
    #z_loc = (random.randint(1, 3), 8)
    #q_loc = (random.randint(6, 8), 2) #rewards only in 2nd half of grid world,␣
 ↪in different columns
    #w_loc = (random.randint(6, 8), 5)
    e_loc = (random.randint(6, 8), 8)
    locs = [y_loc, e_loc]

    gw_array = ['..........', '..........', '..........', '..........', '........
 ↪..',
                '..........', '..........', '..........', '..........', '........
 ↪..',
                ]
    #n*n grid

    s = gw_array[y_loc[0]]
    new = list(s)
    new[y_loc[1]] = 'y'
    s = ''.join(new)
    gw_array[y_loc[0]] = s

    s = gw_array[e_loc[0]]
    new = list(s)
    new[e_loc[1]] = 'e'
    s = ''.join(new)
    gw_array[e_loc[0]] = s



    return gw_array, (locs)

'''

    s = gw_array[x_loc[0]]
    new = list(s)
    new[x_loc[1]] = 'x'
    s = ''.join(new)
    gw_array[x_loc[0]] = s

    s = gw_array[q_loc[0]]
    new = list(s)
    new[q_loc[1]] = 'q'
    s = ''.join(new)
    gw_array[q_loc[0]] = s

    s = gw_array[z_loc[0]]
    new = list(s)
    new[z_loc[1]] = 'z'
    s = ''.join(new)
    gw_array[z_loc[0]] = s


    s = gw_array[w_loc[0]]
    new = list(s)
    new[w_loc[1]] = 'w'
    s = ''.join(new)
    gw_array[w_loc[0]] = s


'''


def create_gridworld(rewards={'.':0, 'y':0, 'e':0}):
    gw_array, locs = create_array() #get the array and the location of goals to␣
 ↪compute fitness
    in_state = (random.randint(4, 6), random.randint(4, 6)) #initial location at␣
 ↪bottom left corner
    gw = GridWorld(gridworld_array=gw_array,
    init_state=in_state, #random initialization of the initial state
    feature_rewards=rewards) #the reward function
    return gw, locs

def learn(gw,m):
    np.random.seed(1234)
    all_run_data = []
    for i in range(1):
        params = {'learning_rate': 1,
              'eligibility_trace_decay': .8,
              'initial_qvalue': 10}
        qlearn = Qlearning(gw,
                   softmax_temp=.2,
                   discount_rate=.99,
                   **params)
        run_data = qlearn.train(episodes=1,
                        max_steps=m,
                        run_id=i,
                        return_run_data=True)
        for r in run_data:
            r.update(params)
        all_run_data.extend(run_data)

    return qlearn, (all_run_data)

def plot_learnt(qlearn, gw): #function to plot the learnt trajectory
    traj = qlearn.run(softmax_temp=0.0, randchoose=0.0, max_steps=1000)
    gwp = gw.plot()
    gwp.plot_trajectory(traj=[(s, a) for s, a, ns, r in traj])

def plot_history(gw, all_run_data):
    run_df = pd.DataFrame(all_run_data) #convert to pandas
    state = run_df["s"] #get states visited
    state_array = state.values #convert to matrix

    action = run_df["a"] #get actions taken
    action_array = action.values #convert to matrix

    gwp = gw.plot() #plot it
    gwp.plot_trajectory(traj=[(s, a) for s in state_array for a in action_array])

def compute_fitness(all_run_data, locs, gw): #function computes fitness over the␣
 ↪history (you can also compute fitness over the learnt policy)
    fitness = 0

    run_df = pd.DataFrame(all_run_data) #convert to pandas
    state = run_df["s"] #get states visited
    state_array = state.values #convert to matrix

    for i in range(len(state_array)):
        a = state_array[i]
        if((9-a[1],a[0]) in locs): #loop through the traj, if traj position is␣
 ↪not same as food position then fitness decreases, else increases by one
            fitness = fitness+1
        else:
            fitness = fitness-0.1
    return fitness
```

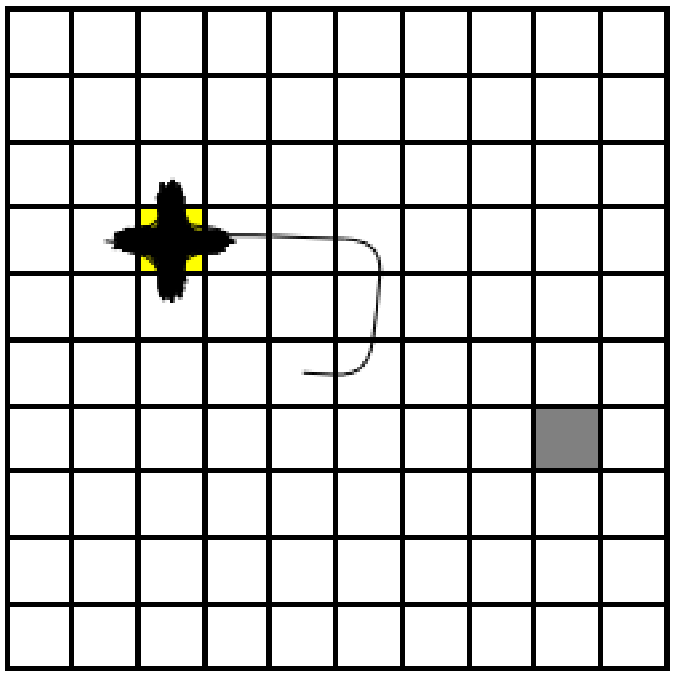
[5]:
```
## REWARDS = 3, Discrete range, all combinations, THOUSAND steps, CENTER initial␣
 ↪state
# Initial state = center
# Episodes = 50
# Trials = 10
# Steps = 1000
# Environments = 10

fitness_vals_thousand = []
y_vals_thousand = []
```

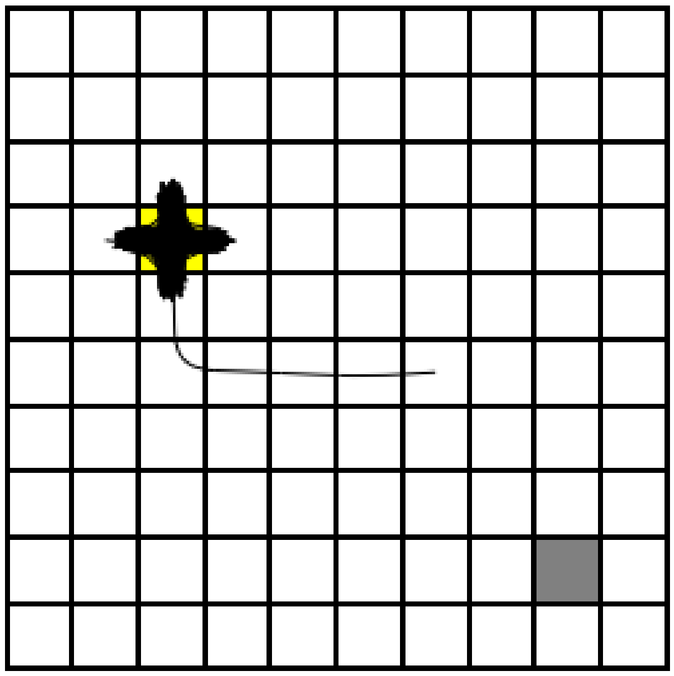
```
e_vals_thousand = []

for y in range(2):
    for e in range(2):
        rewards={'.':0, 'y':y*10, 'e':e*10}
        fit = []
        for env in range(100): #Repeatedly sample lots of environments
            gw, locs = create_gridworld(rewards) #create gridworld here
            qlearn, all_run_data = learn(gw, 10000)     #Q-learn here
            fitness= compute_fitness(all_run_data, locs, gw) #compute fitness␣
 ↪here for the enviroment
            fit.append(fitness)
            if (env == 1):
                print("Rewards are set as:")
                print('y =', y, 'e =', e)
                plot_learnt(qlearn, gw)


        fitness_vals_thousand.append(sum(fit)/float(len(fit)))
        y_vals_thousand.append(y)
        e_vals_thousand.append(e)
        print("Average Fitness", sum(fit)/float(len(fit)))
```

```
Rewards are set as:
y = 0 e = 0
Average Fitness -741.7860000001316
Rewards are set as:
y = 0 e = 1
Average Fitness 3218.7310000003013
Rewards are set as:
y = 1 e = 0
Average Fitness 3368.0340000003275
Rewards are set as:
y = 1 e = 1
Average Fitness 3396.436000000332
```



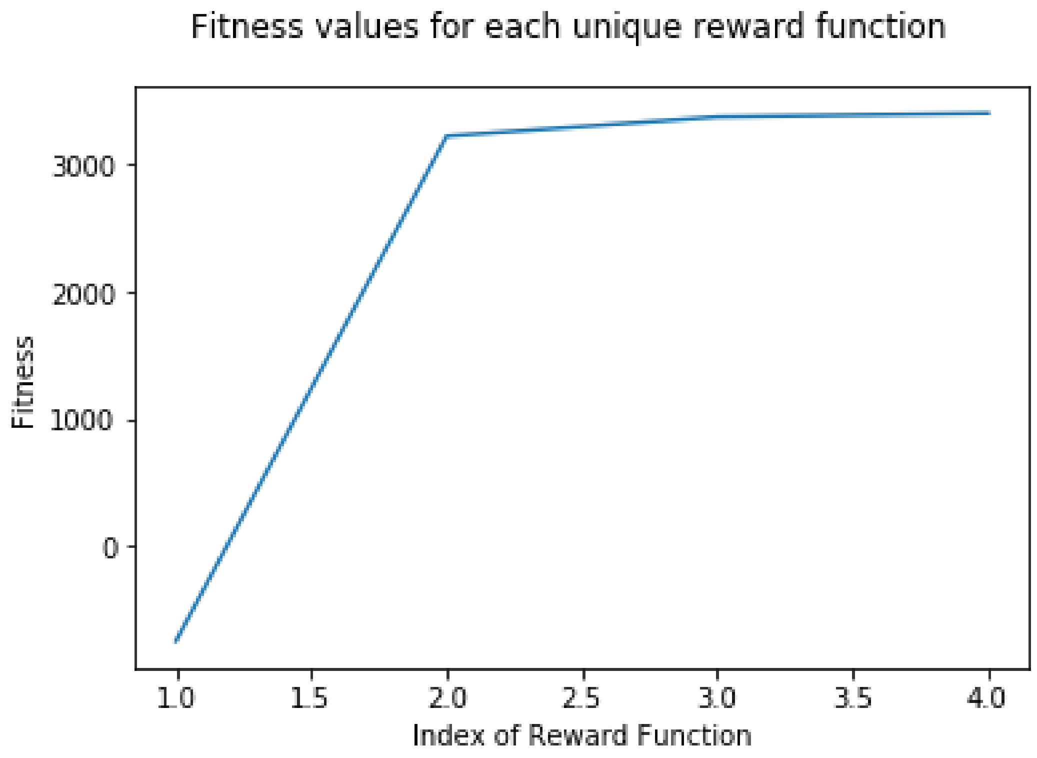






[11]:
```
fig, axis = plt.subplots()

axis.plot(np.arange(1,5),fitness_vals_thousand)

fig.suptitle("Fitness values for each unique reward function", fontsize=12)
axis.set_xlabel('Index of Reward Function', fontsize=10)
axis.set_ylabel('Fitness', fontsize=10)

mean_center_init = np.mean(fitness_vals_thousand)
variance_center_init = np.var(fitness_vals_thousand)
cov_center_init = (variance_center_init/mean_center_init)*100

print("Mean Fitness over all reward functions = ", mean_center_init)
print("Variance of Fitness over all reward functions = ", variance_center_init)
print("Coefficient of Variation over all reward functions = ", cov_center_init)

mean_center_init = np.mean(fitness_vals_thousand[1:])
variance_center_init = np.var(fitness_vals_thousand[1:])
cov_center_init = (variance_center_init/mean_center_init)*100

print("Mean Fitness over all reward functions except case 1 with all rewards set␣
  ↪0 = ", mean_center_init)
print("Variance of Fitness over all reward functions except case 1 with all␣
  ↪rewards set 0 = ", variance_center_init)
print("Coefficient of variation over all reward functions except case 1 with all␣
  ↪rewards set 0 = ", cov_center_init)
```

```
Mean Fitness over all reward functions =  2310.3537500002076
Variance of Fitness over all reward functions =  3109742.111713879
Coefficient of Variation over all reward functions =  134600.25815153198
Mean Fitness over all reward functions except case 1 with all rewards set 0 =
3327.733666666987
Variance of Fitness over all reward functions except case 1 with all rewards set
0 =  6075.236270891008
Coefficient of variation over all reward functions except case 1 with all
rewards set 0 =  182.5637770157214
```


Fitness values for each unique reward function
Fitness
3000
2000
1000
0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Index of Reward Function

```
[ ]:
```