Reinforcement learning plan.

The current approach to team allocation uses a numerical optimization routine based on Scipy's *basin-hopping* with linear optimization conducted on each hop. This approach is slow and the numerical optimization must be run for every project meaning that 'watching' simulations unfold in real-time is not feasible.

A reinforcement learning approach could improve on these run-times by separating out the two phases of training and inference. Essentially the training stage would be done up front and would be (potentially very) slow. During training the reinforcement learning (RL) algorithm would learn about the problem of team allocation by *exploring* the associated Markov Decision Process (MDP). If successful, the training will learn an *optimal policy* for team allocation i.e., a function that maps *states* (project requirements + workforce parameters) to *actions* (sequential allocation of workers). The policy will be optimal in the sense that it provides the action that maximizes the reward for any given state[1]. Once this policy has been learned, it should then be much quicker to conduct inference i.e., to use the policy to determine optimal team allocation for any given project and workforce condition.

In order to pose the task of team allocation as an RL problem, we need to define the system as an MDP where actions (**a**) map states (**s**) to new states (**s'**) plus a reward (**r**). The reward **r** is given by a function **r = R_a(s, s')**, which indicates the reward given when playing action **a** in state **s** and moving to state **s'**:

s —a—> r, s'

The mapping can be probabilistic, but in our case is deterministic which is simpler: from a given state **s**, taking action **a** moves the system to a known new state **s'** with probability 1**.**

Therefore, we need to define the following:

- **States**: captures the whole state of the system as a *state vector.* Full project definition and workforce state including history.
- **Actions**: the action space is the set of all actions that can be taken (not all actions will be valid for any given state). In our case a single action is the assignment of a single worker hard skill unit to a project.
- **Reward**: the reward function **R_a(s, s'),** which in our deterministic case reduces to **R_a(s)**. This is the value that we are trying to maximize and in our case is simply the probability of project success for the current skill unit assignments.

The task of RL is to find a **policy π: S -> A** that maps states to actions such that the expected return (sum of the rewards) is maximized.

In the case where the state vector and the action space are finite, we can use classical reinforcement learning where the dynamic programming algorithms *value iteration and policy iteration* give optimal policy solutions. This is known as the **tabular setting.** In reality we need the system not just to be finite, we need to be able to represent every state-action pair in a table that will fit in the memory of a computer. In other cases, an exact solution cannot be found because the space is infinite or the MDP is poorly specified. In these cases, deep-learning (neural networks) can be used to approximate a solution

[1] Note that optimal result is only guaranteed in the 'tabular' or finite case. When the problem size is too large we must use function approximation (neural networks) and convergence to an optimal solution is not guaranteed.

and this can work very well given sufficient data or simulation time. It is likely that we will either need to make some simplifications/approximations so that we can work in the tabular setting, or we will need to use deep-learning to find approximate (but very good!) solutions to the team allocation problem. **See suggested approach below.**

Estimating the dimensionality of our state-action space.

We define the following values:

H: number of hard skills (default = 5)

F: number of soft skills (default = 5)

W: number of workers (default = 100)

S_{min}: minimum skill value

S_{max}: maximum skill value

From which it follows that the size of the finite action space is **|A| = W*H** because this is the number of distinct actions that we can choose between in assigning worker hard skill units to a project.

The state vector has the following components:

Component	Variable	Symbol	Dimension	Value space	Approximate value space	Number of discrete states
Project	Skill requirements	(L_i, U_i) for i = 1,..H where L_i and U_i are the level and unit requirements of hard skill i	2H	$\mathbb{R}^+$ (<5)	~~Discretized to 1 d.p. on [0,5]~~ Discretized to nearest integer on [0,5]	2H * 5
~~Project~~	~~Risk~~	~~r~~	~~1~~	~~5,10,25~~	~~N/A~~	~~3~~
Project	Creativity	c	1	1,2,3,45	N/A	5
~~Project~~	~~Budget~~	~~b~~	~~1~~	~~$\mathbb{R}^+$ (<175)~~	~~Discretized to nearest 10 on [0,180]~~	~~18~~
Workforce (For each of W workers)	Hard skills	s_i for i = 1,..H	H	$\mathbb{R}^+$ (<5)	~~Discretized to 1 d.p. on [0,5]~~ Discretized to nearest integer on [0,5]	H * 5 Per worker
Workforce (For each of W workers)	Soft skills	k_i for i = 1,..F	F	$\mathbb{R}^+$ (<5)	~~Discretized to 1 d.p. on [0,5]~~ Discretized to nearest integer on [0,5]	H * 5 Per worker
Workforce	Momentum	m	1	0,1	N/A	2 Per worker

(For each of W workers)						
Workforce	Social network	G	W(W-1)/2	$\mathbb{N}^{W(W-1)/2}$	N/A	W(W-1) [2]
Team	Skill allocation vector	V	W*H	0,1	N/A	2*W*H

To be able to use the exact solutions for policy or value iteration. We require that the state vector is of finite dimension and that each element of the state vector can only take a finite number of values. This means that there are a **finite number of discrete states**. We can achieve this by simple approximation that should not impact performance significantly (e.g. rounding real-numbered skill values to the nearest decimal place).

Based on the above table, we have 2H*5 * 5 * WH*5 * WH*5 * 2W * W(W-1) * 2*W*H discrete states in total. With W=100 and H=5, this gives **3.1 x 10^{18} states!**

This is far too many states for the tabular methods, as they won't fit in memory. And even if they did, the training would be too slow (years or decades). So, we will need to use some combination of simplification and deep-learning methods.

Suggested plan of action:

1. Break the team allocation tasks in smaller component tasks based on the probability function, so that these can be solved in parallel by separate RL agents (at least initially).
2. Use tabular methods (finite solution) to find optimal policy for small numbers of workers (e.g. 5, 10, 20). Determine how run-time and memory usage is scaling with number of workers. Compare performance to basin-hopping.
3. Use neural network (function approximation) methods to find the solution to the same problems.
4. Compare performance of NN solutions against tabular methods and basin-hopping optimization. Explore stability of NN solutions.
5. Scale NN to full problem (100 workers). Conduct same performance and run-time comparisons against basin-hopping.

[2] This is because we only need to use a binary (unweighted) version of the social network for calculation of project success probability, since the number of times a pair of workers have successfully collaborated in the past does not currently impact the probability calculation.

Notes and ideas for development....

Note: These calculations assume 100 workers and 5 hard and soft skills, which are the current default parameters. The RL methodology developed here is for organizations with fixed numbers of workers – the algorithm would need to re-trained to account for changes in the number of workers. And the training time will scale non-linearly with the number of workers. Experiments will determine the scaling. One solution with a larger number of workers (say an organization of 1000 workers), would be to pre-filter the workers to select a pool of applicants based on some criteria and then use the same RL algorithm that was trained for 100 workers.

Ideas on how to reduce dimensionality further:

- Reframe the problem as choosing workers, and let the skill assignment happen 'automatically'?
- There are constraints on the allocation of workers that limits the viable action space for a given state. This should reduce the amount of exploration that is required during training. For example, there is a limit of 7 workers in a team, so not all allocation vectors are viable. Similarly, the budget constraint reduces the number of viable allocation vectors.
- We can scrap the budget contribution because the budget is defined by the project skill requirements (duplication in representation).
- Can remove risk components because it is a constant so does not affect the optimisation.
- Could scrap soft skills or other less important components of probability function (to get an approximate solution?)
- **Could scrap social network storage (include history in the game playing i.e. MDP to include multiple projects in sequence).**
- **Treat as parallel smaller component problems: find high creativity teams, find high OVR teams, find good skill balance teams...i.e. solve for components of probability function. Then search the solution space for shared teams. (Hash the teams for quick search/comparison - if no match, compute creativity scores for top X teams etc.)**

Efficiency and optimization considerations for RL:

- Keep S-A pairs in hash table for fast lookup
- Using after-state values instead of action values for deterministic MDPs
- Deep Q learning tutorial: pythonprogramming.net
- GPU accelerated implementation of policy iteration in Python?

References:

1. "All of these algorithms converge to an optimal policy for discounted finite MDPs" see http://incompleteideas.net/book/ebook/node44.html
2. Discussion on state space size: https://ai.stackexchange.com/questions/24883/what-constitutes-a-large-space-state-in-q-learning
3. Tabular RL methods: https://towardsdatascience.com/summary-of-tabular-methods-in-reinforcement-learning-39d653e904af
4. What to do when too many states: https://www.quora.com/How-do-you-apply-Q-learning-when-there-are-too-many-possible-states-and-therefore-you-dont-have-enough-memory-to-

write-and-update-the-q-table-Ex-I-dont-have-enough-memory-for-chess-q-table-How-do-you-deal-with-it

5. After-state values (not action values): https://stats.stackexchange.com/questions/411932/reinforcement-learning-afterstate-and-afterstate-value-functions
6. Reachable states: https://ai.stackexchange.com/questions/19856/non-neural-network-algorithms-for-large-state-space-in-zero-sum-games