**Definition**

**State**: Each state represents the current column value (i.e., the result after updating B\_old by taking the selected actions).

**Action List**: under current state B\_cur, the candidate operations (consisting of an 'add' transformation and a 'range' predicate).

**Reward**: the evaluation of the chosen action's effect on the **environment** (environment in our problem means the path from B\_cur to B\_new, and the path means the selected action list).

**Target**: the final state, either B\_new or the state whose seletectd # of operations > the # of operations returned by the 2-approx algorithm;

**State Tree**: a modified Trie Tree. Each node in this tree represents a state, and each child node of it means a possible action the model could choose to operate.

**Background: Reinforcement Learning**

In the Reinforcement learning model, more specifically the Policy Gradient model I choose, the assumption is that in real world people do not know the effect of each action until they get the final outcome. They may not be able to label each action as good or bad at the current state but they may have the **reward** which signifies the action’s effect.

The RL model uses a **batch** as a train unit. Within each **batch**, the model starts from the start state. To expand the current state, users need to provide the **action list**. Users also need to define **a feature vector** for each action so that these feature vectors can be applied as the input of the **policy network** (the dimension of the Network input is fixed in the model). The RL model applythe policy network to output the **probability distribution** ***p*** (is a parameter), the chance of choosing each action. Based on ***p***, the model chooses one action and moves to the corresponding next state. Meanwhile, this next state receives a **reward** (user defined) as ***rt***at this time step **t** (and the **discount reward** is ***Rt*** = (γk) \* *rt+k*, y is a number between 0 and 1 called a discount factor). Through the state expanding, the procedure finally reach target, which signifies the end of a **batch**. At the end of each batch, we can calculate the expected value function ***J*** as ∑ ***pt \* Rt*.** Then after 100 (defined by us) batches (i.e., a epoch), the model applies gradient method for updating to optimize the ***J.***

**Model**

In our problem, we can define each **state** as B\_cur.

The **action list** for each state is based on the previous assumption: if we focus on the ‘A-B +’ case, the tuples in B can be **sorted** by the value of A. In this way, we do not need to consider A since the consecutive tuples in B signify the possible ‘**range predicate**’ in A. Also in this setting the order of operations means that some tuples can be updated more than once. The ideal strategy is we can detect the ‘update once’ operation which performed on the consecutive tuples which were updated once, then detect the ‘update twice’ operation. But what we actually can detect based on B\_cur and B\_new is the **separated blocks** of B, each block contains a consecutive tuples which were updated by the same transformation (i.e., add the same value). These **transformations** should be the all possible transformations at B\_cur to B\_new, since it is weird if we perform one operation but there are no tuples reach B\_new. And the ‘**range predicate**’ can be the combination of these blocks. Under this assumption, we limit the search space by constraining the action list as above. Moreover, since the dimension of policy network is fixed, we need to randomly sample 20 (defined by us) actions from the action list as the input of the policy network.

The **feature function** is defined by us which contains five dimensions (which should be normalized between 0-1):

1. B\_cur / (# of different transformations at B\_old);

# of different transformations at B\_cur

2. the B\_new / (total number of tuples)

# of tuples at the current state that is different from

3. np.abs(cur\_dis - ori\_dis) / ori\_dis

(dis is the distance between B\_cur and B\_new, ori\_dis is the distance between B\_old and B\_new);

4. (greedy\_op + depth + 1) / ori\_op

(ori\_op: 2\*approx number of operations, greedy\_op: calculated by the greedy algorithm that the predicated operations to transfer B\_cur to B\_new);

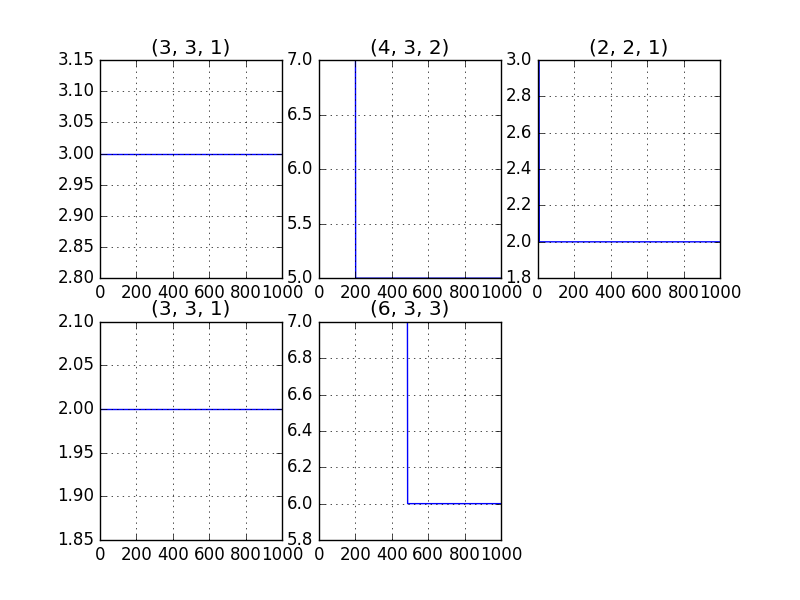
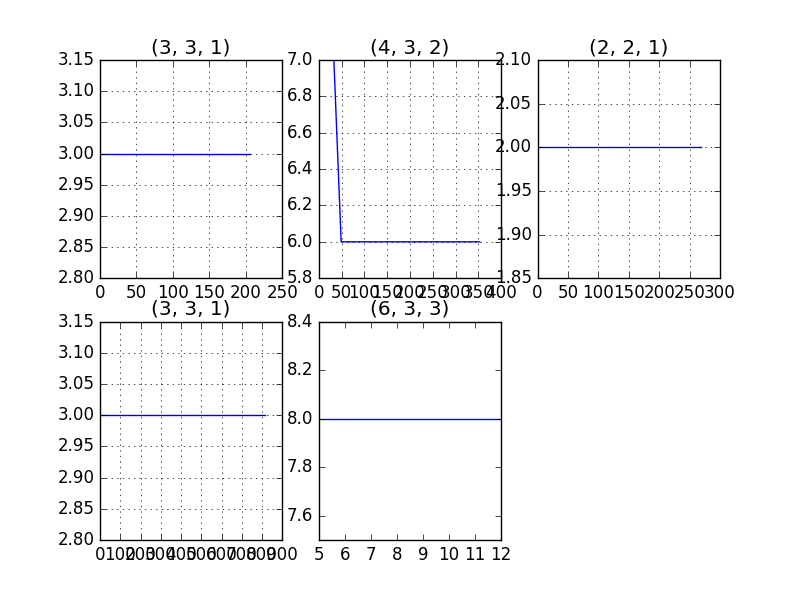
5. (depth+1) / min\_op

(min\_op is the current minimum operations to transfer B\_old to B\_new)

The **reward strategy** is also define by us as 1. If the number of operations chosen at this batch is greater than min\_op, then reward strategy =-1000; 2. elif B\_cur == B\_new, then the reward strategy = ori\_op / depth (depth: the number of operations this batch choosed)\*10000; 3. elif B\_cur != B\_new, then the reward strategy = ori\_op / (depth + greedy\_op) \* 1000.

Moreover, we balance the **exploring and exploiting** strategy when expanding the state. Within each batch, if we choose the exploiting strategy, the model starts from the B\_old and then reaches the target state; on the other hand, if we choose the exploring strategy, we maintain a state tree so that we can backtrace a fixed number (defined by us) of other possible actions in the past state when the model reaches the target state. We also add a **replay pool** which contains several previous trained dataset selected by a forgetting rate. Moreover, in order to fix the overfitting problem, at each state, we also use the \**epsilon-greedy** strategy to select either the predicted action returned by the model or a randomly selected action.

We also show the experimental results below. The left set of charts represents the results from RL model, while the right set of charts represents the searching result.



However, for the search algorithm, if we replace the recursive strategy with a stack that memorizes the results of the searched state, the searching algorithm can ensure to find the best answer within a memory space limitation. But if we use say 1000 batch heuristic searching result as the label to supervise the training of the RL model, we can ensure that the RL model performs better than the search algorithm.

**Discussion**

Some drawbacks in this model are listed as following:

* The definition of feature. We want to make this model be more general but the variance between each dataset is large. So the feature we defined may not be representative for a particular action.
* The sampling strategy. Since the input dimension of policy network is fixed, it might be not wise to randomly sample 20 actions as the input for the policy network.
* The exploring strategy. In the searching algorithm, we can backtrace any node in the state tree whose weight is less than the current selected node. We may also want to maintain a global structure to record the reward for each state in the RL model for backtrace.

There are two most important obstacles in this problem. First of all, for the linear regression case, we may not know whether a operation is good or bad for the environment. In the ‘AB +’ case, we can defined clearly that reducing blocks is better than not. But in the ‘AB linear transformation’ case, we could not clearly define the ‘blocks’. Therefore, it constrains the definition of the feature vector, the weight and the reward. Another obstacle is related to the action list. In the ‘AB +’ case we can limit the searching space by the assumption we talked earlier. If we consider the multiple A case, we can not clearly constrain the possible range predicate of several A but rather need to test each range of A (this would give us huge search space). More specifically, it is the information (B\_cur, B\_new, multiple A) we have that may not be able to help us determine which action is not necessary; thus we must search each action.