BlueYonder

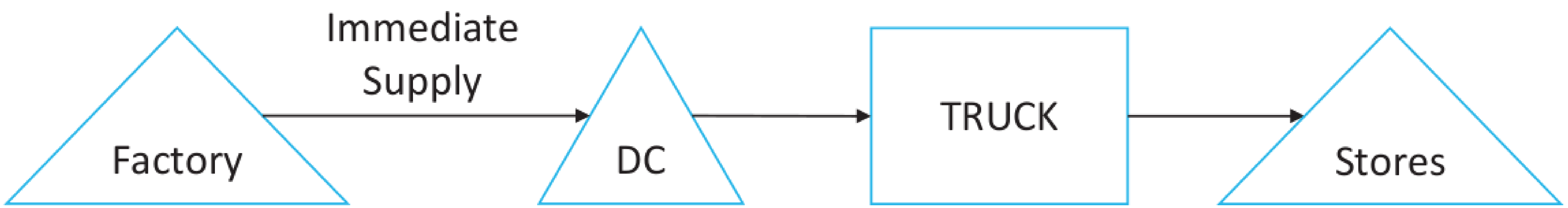
Fulfill your potential™

DEMAND AND SUPPLY PROBLEM

THE PROBLEM STATEMENT

Develop a Demand and Supply algorithm

Planning Horizon: 3 Months | Supply Chain Example



Daily Capacity = 200 units per Day.
Cannot store anything here.

Simulate for Storage Capacity of 50, 100 and 200 units

Per Truck Capacity = 20 units

Sample demand with different Probability distributions between 10 and 400 per day per product

Problem:

- Develop an algorithm to compute the minimum number of Trucks needed to satisfy the maximum demand given the Capacity and Storage Constraints.

Note: Depending on the demand variability, you would need to store stock at DC by building ahead of time at the factory. On the other hand, you need trucking capacity to move the Stock to the Stores in case it exceeds Storage capacity at DC.

Registration Link: Innovation with Academia event.

INPUTS

- Demand range
- Set of stores
- Set of Products
- Time frame
- DC
- Truck capacity
- Factory Production Limit
- Demand Probability Distribution
- Current Demand

INPUTS

- Demand range : [10, 400]
- Set of stores : S1, S2…Ss
- Set of Products : P1, P2…Pp
- Time frame : [0,90)
- DC : capacity(50,100, 200) and current state(Pi->N)
- Truck capacity : 20
- Factory Production Limit : 200
- Demand Probability Distribution : Stores x Products x Time Frame x Demand range -> [0,1]
- Current Demand : Store x Product -> Demand Range

ASSUMPTIONS

- The factory produces products instantly
- The factory to DC transport is instant.

DEFINITIONS

- Episode : an episode is the actions and states encountered to reach the goal state. Our case its 90 days of demands and our responses to those demands

REINFORCEMENT LEARNING

Environment

Demand Probability Distribution

Goal

The end of the Timeframe (90 days) period

Agent

Us.

State

Current cumulative demand and semi filled trucks list

Actions

Deciding which trucks not to send, and what products to omit, as well as which products to stock the DC

Reward

-1 x Wastage in trucking *

EXPECTIMAX TREE

2 STORES, 2 PRODUCTS

Number of possible demands we can receive

$Demand\ Range^{|products|*|stores|}$ = 2.3 * 10e11

Number of choices we have to make

Choice 1 : If factory and DC together cannot meet the current demand, which overflow products are we dropping

$^{Factory\ capacity+|product|*|store|-1}C_{Factory\ capacity}$ = 1373701

Choice 2: For each store, drop the (possibly) semi filled truck going there

$2^{|stores|}$ = 4

Choice 3: How much of the DC we restock and with what

$^{DC\ capacity+|product|}C_{DC\ capacity}$ = 70058751

At depth 6

= 4.05 x 10e84

MONTE CARLO TREE SEARCH

ADVANTAGES

- Always Ready with an answer
- Only one function is required to work i.e. additional constraints can be easily added (MyAction Node::createRandomMyAction)
- One function calculates fitness i.e Heuristic Intelligence can be easily added
 (float Node::evaluate)
- Trivially parallelizable

IMPLIMENTATION

Abstract Base Node Class Maintains
- parent Node
- depth
- number of simulations
- fitness accumulated via simulations

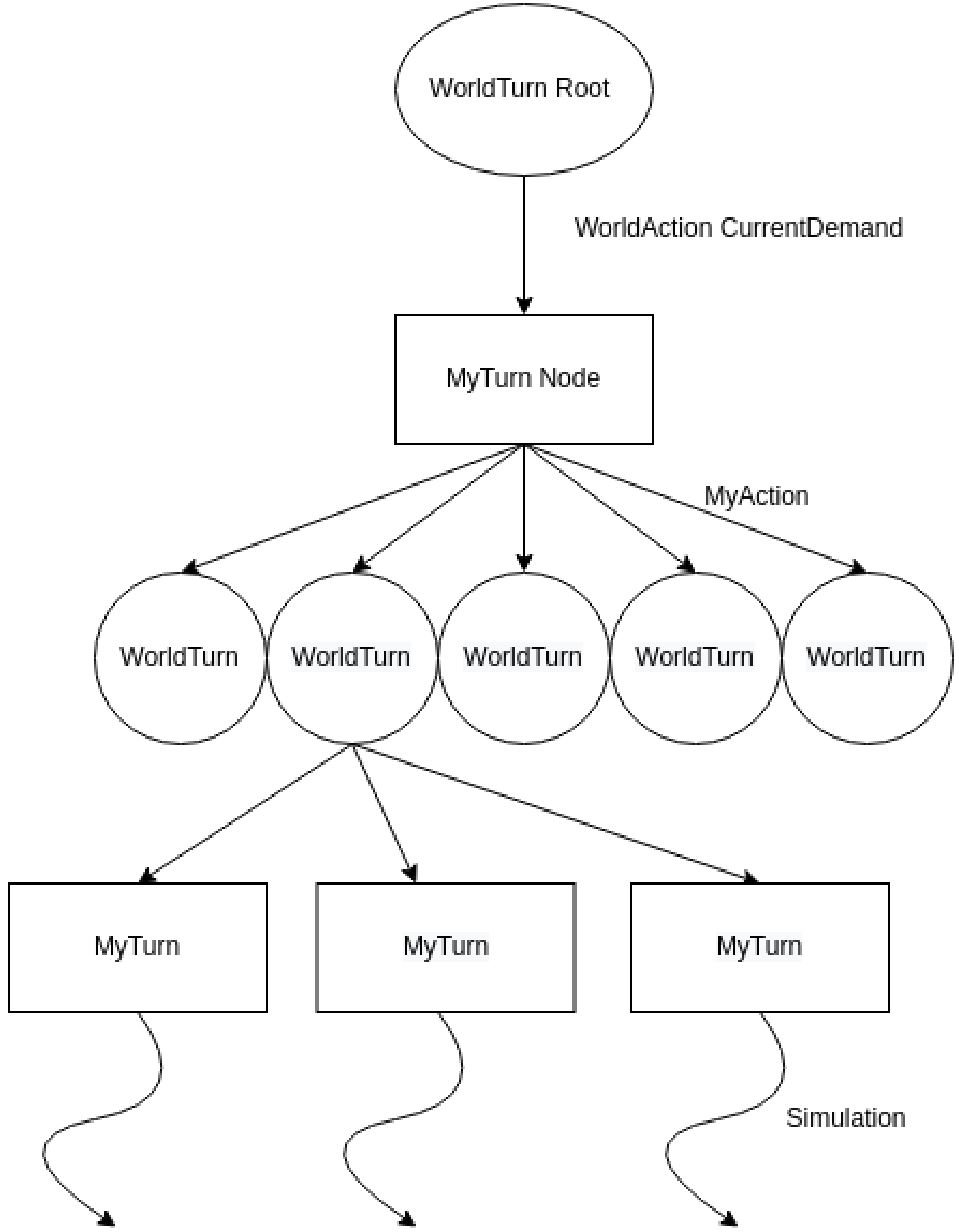
World Action Class
- demand

MyAction Class
- Factory Produce
- Trucks

WorldTurn Node
- derives Node class
- has MyTurn Nodes with WorldAction Edges

MyTurn Node
- derives Nodes
- has WorldTurn children with MyAction Edges
- Simulations are run from these nodes

Monte Carlo Tree Search Class:
- has WorldTurn root node
- executes the MCTS main loop



```cpp
struct WorldAction{
  std::vector < std::vector < int > > demand; // [store->[proudct -> demand]]
  WorldAction(int nmbr_strs, int nmbr_prdcts) :demand(nmbr_strs, std::vector<int>(nmbr_prdcts)) {};
  WorldAction(const std::vector<std::vector<int> > &data_) : demand(data_){};
};
struct MyAction{
  std::vector < std::pair< int, std::vector < int > > > trucks; //[truck->(store, [ product_id -> amount ])]
  std::vector<int> total_prduc; //[product -> total amount(includes whats stocked and what sent in trucks)]
};
class Node{
protected:
  Node* parent_;
  float success_;
  int nmbr_simls_;
  int depth_;
  virtual bool IamLeaf() = 0;
  Node(Node* parent, int depth): parent_{parent}, success_{0}, nmbr_simls_{0}, depth_{depth} {};
public:

  float fitness(int parent_smls, float exploration_factor);
  void backpropagate(float fitness);

  virtual Node* select( float exploration_factor ) = 0;
```

```cpp
class WorldTurn: public Node{
private:
  std::list<std::pair<WorldAction, MyTurn*> > children;
```

```cpp
class MyTurn : public Node{
private:
  std::list<std::pair<MyAction, WorldTurn*> > children;
```

```
static WorldAction createRandomWorldAction(int nmbr_strs, int nmbr_prdcts, int time,int time_frm, int demand_range,
static void createRandomMyActionSubRoutine_GenerateTuckingForStore(const std::vector<bool> &willLastTruckBeSent, co
static int createRandomMyActionSubRoutine_LiabilityTrkVc(std::vector<bool> &willLastTruckBeSent, const std::vector<
static MyAction createRandomMyAction(int truck_capacity, int factory_production_limit, int DC_cpcty, const std::vec
static float evaluate(const std::vector<std::vector<int> > &crnt_total_demand, const std::vector<std::vector<int> >
```

```
class MonteCarloTreeSearchCpp {
private:
  WorldTurn* treeRoot;
```

```
void MonteCarloTreeSearchCpp::MainLoop () {
  // std::cout<<"Starting select"<<std::endl;
  WorldTurn* node = static_cast<WorldTurn* >(treeRoot->select(this->exploration_factor_));
  // std::cout<<"Starting Expand"<<std::endl;
  node->expand( this->nmbr_brnch_wrldTurn_, this->nmbr_brnch_myTurn_, this->wareHouseState_,this->cmltv_demand_prb_dstrbutn_, this->nmbr_strs_,
    this->nmbr_prdcts_, this->time_frm_, this->demand_range_, this->demand_min_ , this->truck_capacity_, this->factory_production_limit_,
    this->DC_cpcty_, this->nmbr_simulations_per_rollout_, this->trk_thrpt_rti_cnst_);
}
```

IMPROVEMENTS

1. Write CUDA code to parallelize the simulation process

IMPROVEMENTS

1. Write CUDA code to parallelize the simulation process

2. Create a Graphics Interactive Interface

IMPROVEMENTS

1. Write CUDA code to parallelize the simulation process

2. Create a Graphics Interactive Interface

3. Shift the entire tree to GPU and make it parallel across GPU cluster

IMPROVEMENTS

1. Write CUDA code to parallelize the simulation process

2. Create a Graphics Interactive Interface

3. Shift the entire tree to GPU and make it parallel across GPU cluster

4. Whatever you want!

Thank you!