Anytime, Forward-Searching, Depth-Bounded, Utility Driven Scheduler for State Space Simulation

Henry Gilbert

Introduction

State Quality Function

Modified Search Function

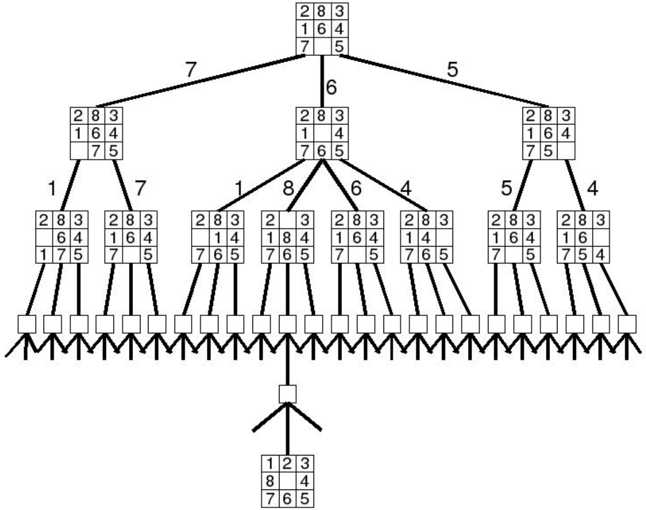
Resources

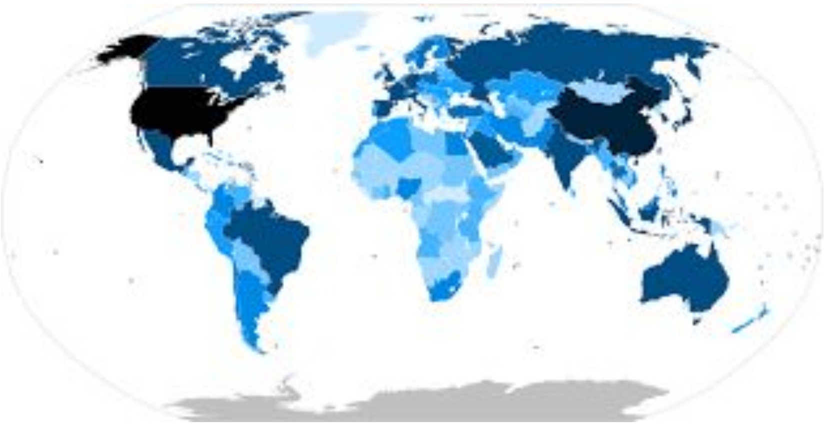
Transformations

Trades

Experiments

Conclusion





Sequential Search Function

Modeled off given search function

Utilizes custom priority queue based off of solution EU

Depth bounded by parameter For each new solution, calculate possible actions

```
def search(self):
    """ This is the generic anytime, forward searching, depth-bound,
    generic utility driven scheduler as outlined in the slides. Given a new state,
    all possible next states are computed and then sorted based on EU. The highest
    state is poped from the queue until the queue is empty and only solutions remain.

    Arguements:
        None

    Returns:
        None
    """

    initial_solution = Solution(self.country.state_value(),
        [[None, self.country, self.countries, 0]])

    self.frontier.push(initial_solution)

    while not self.frontier.empty():

        solution = self.frontier.pop()

        if len(solution.path) > self.depth:
            self.solutions.push(solution)
            continue

        self.generate_succesors(solution)
```

Parallel Search Function

Same process as sequential until inflection point

Frontier is chunked by number of cpus and paralized

States are now processed as a beam search, until the given depth is reached

Resulted in exponential temporal improvement vs sequential

```python
def search_parallel(self) -> None:
    """This function searches the possible state space in parallel,
    utilizing all possible cores on a given machine. Once the number
    of searchable states hits the infliction point, they are chunked,
    and parallized, where beam search is then used to search to the
    given depth. The results are returned and sorted in the priority queue.

    Arguements:
        None

    Returns:
        None
    """

    total = 0
    initial_solution = Solution(self.country.state_value(), [[None, self.country, self.countries, 0]])
    self.frontier.push(initial_solution)
    seg_num = int(self.max_frontier_size/os.cpu_count())

    while not self.frontier.empty():

        if len(self.frontier.queue) == seg_num:      # Maybe we only need the top 10?

            print("starting parallel")

            shared_frontier = mp.Manager().list()
            pool = mp.Pool()
            chunks = np.array_split(np.array(self.frontier.queue), os.cpu_count())
            self.frontier.queue = []

            for chunk in chunks:
                pool.apply_async(
                    func=generate_succesors_parallel,
                    args=(chunk, self.countries,
                          shared_frontier, self.gamma,
                          self.r_weights, self.state_reduction,
                          self.depth, seg_num)
                    )

            pool.close()
            pool.join()

            print("out of parallel")

            total += len(shared_frontier)
            for sol in shared_frontier:

                if len(sol.path) > self.depth:
                    self.solutions.push(sol)
                else:
                    print("Shouldn't be in here")
                    self.frontier.push(sol)

        else:

            solution = self.frontier.pop()
            total += 1

            if len(solution.path) > self.depth:
                self.solutions.push(solution)
                continue

            self.generate_succesors(solution)
```

Transfer Solution – P1

Capturing the current state and environment

Creating storage dictionaries to house the amount of tradeable elements

Same process is repeated for every other country

```python
def generate_transfer_succesors(self, solution: Solution):
    """Given the current solution, computes all equal trades
    between all countries and all resources. Stores corresponding
    new states and transfer in the given frontier.

    Args:
        solution (Solution): Current solution

    Returns:
        None
    """

    curr_state = solution.path[-1][1]
    curr_countries = solution.path[-1][2]
    countries_elms = {}

    curr_elms = {
        'metalic_elm': curr_state.metalic_elm,
        'timber': curr_state.timber,
        'available_land': curr_state.available_land,
        'water': curr_state.water,
    }

    for c in curr_countries:
        countries_elms[c] = {
            'metalic_elm': curr_countries[c].metalic_elm,
            'timber': curr_countries[c].timber,
            'available_land': curr_state.available_land,
            'water': curr_state.water,
        }
```

Transfer Solution – P2

Calculates relative resource worth

Calculates all possible relative equal value trades

Excludes redundant or same element trades for greater efficiency

Grouping and averaging of solutions based off given state reduction hyperparameter

Simulates and stores into frontier

```python
for c in countries_elms:
    for elm in countries_elms[c]:

        for curr_elm in curr_elms:

            if curr_elm == elm:     # Skipping to avoid redundant trades
                continue

            other_elm_scale = 1 / self.r_weights[curr_elm]
            self_elm_scale = 1 / self.r_weights[elm]
            max_amount = min(int(countries_elms[c][elm]/other_elm_scale), int(curr_elms[curr_elm]/self_elm_scale))

            if max_amount <= 0:
                continue

            if self.state_reduction == -1:       # Ultimate reduction

                other_elm_amount = ceil(max_amount / other_elm_scale)
                self_elm_amount = ceil(max_amount / self_elm_scale)

                trade = Transfer(elm, curr_elm, other_elm_amount, self_elm_amount, c, curr_state.name)
                new_curr_state = curr_state.make_trade(curr_elm, self_elm_amount)
                new_countries = copy.deepcopy(curr_countries)
                new_countries[c] = curr_countries[c].make_trade(elm, other_elm_amount)
                new_solution = copy.deepcopy(solution)
                new_solution.path += [[trade, new_curr_state, new_countries]]
                self.calculate_reward(new_solution)
                self.frontier.push(new_solution)

            else:

                poss_trades = [i+1 for i in range(max_amount)]
                num_buckets = ceil(len(poss_trades) / self.state_reduction)

                if num_buckets < 1 or len(poss_trades) == 0:
                    continue

                amounts = []
                buckets = np.array_split(poss_trades, num_buckets)

                for bucket in buckets:
                    if len(bucket) > 0:
                        amounts.append(int(sum(bucket)/len(bucket)))

                for amount in amounts:

                    other_elm_amount = ceil(amount / other_elm_scale)
                    self_elm_amount = ceil(amount / self_elm_scale)

                    trade = Transfer(
                        elm, curr_elm, other_elm_amount, self_elm_amount, c, curr_state.name)
                    new_curr_state = curr_state.make_trade(
                        curr_elm, self_elm_amount)
                    new_countries = copy.deepcopy(curr_countries)
                    new_countries[c] = curr_countries[c].make_trade(elm, other_elm_amount)
                    new_solution = copy.deepcopy(solution)
                    new_solution.path += [[trade, new_curr_state, new_countries]]
                    self.calculate_reward(new_solution)
                    self.frontier.push(new_solution)
```

Transform Solution

Calculates all possible transformations

Simulates and stores all transformations

```python
def generate_transform_succesors(self, solution: Solution):
    """Given the current solution, computes all possible transforms
    and the resulting states. Stores new states and corresponding
    transforms in the frontier.

    Args:
        solution (Solution): Current solution

    Returns:
        None
    """

    curr_state = solution.path[-1][1]

    housing_scalers = curr_state.can_housing_transform()
    alloy_scalers = curr_state.can_alloys_transform()
    electronics_scalers = curr_state.can_electronics_transform()
    food_scalers = curr_state.can_food_transform()
    farm_scalers = curr_state.can_farm_transform()

    for scaler in housing_scalers:

        trans = HousingTransform(scaler)
        new_state = curr_state.housing_transform(scaler)
        new_solution = copy.deepcopy(solution)
        new_solution.path += [[trans, new_state, self.countries]]
        self.calculate_reward(new_solution)
        self.frontier.push(new_solution)

    for scaler in alloy_scalers:

        trans = AlloyTransform(scaler)
        new_state = curr_state.alloys_transform(scaler)
        new_solution = copy.deepcopy(solution)
        new_solution.path += [[trans, new_state, self.countries]]
        self.calculate_reward(new_solution)
        self.frontier.push(new_solution)

    for scaler in electronics_scalers:

        trans = ElectronicTransform(scaler)
        new_state = curr_state.electronics_transform(scaler)
        new_solution = copy.deepcopy(solution)
        new_solution.path += [[trans, new_state, self.countries]]
        self.calculate_reward(new_solution)
        self.frontier.push(new_solution)

    for scaler in food_scalers:

        trans = FoodTransform(scaler)
        new_state = curr_state.food_transform(scaler)
        new_solution = copy.deepcopy(solution)
        new_solution.path += [[trans, new_state, self.countries]]
        self.calculate_reward(new_solution)
        self.frontier.push(new_solution)

    for scaler in farm_scalers:

        trans = FarmTransform(scaler)
        new_state = curr_state.farm_transform(scaler)
        new_solution = copy.deepcopy(solution)
        new_solution.path += [[trans, new_state, self.countries]]
        self.calculate_reward(new_solution)
        self.frontier.push(new_solution)
```

Computing Potential Transforms

Calculates maximum scaler factor for potential transform based off base requirements

Computes all possible transform factors

Chunks solutions based off initiate state reduction factor

```
def can_food_transform(self):
    """Function to calculate possible scalers for a
    housing transform given the current state of the country.

    Parameters:
        None

    Returns:
        list: List of potential scalers for a housing transform.
    """

    if (self.farm >=5 and self.water >= 10):

        scalers = []
        scalers.append(int(self.farm / 5))
        scalers.append(int(self.water / 10))

        if self.state_reduction == -1:
            return [min(scalers)]

        poss_scalers = [i+1 for i in range(min(scalers))]
        num_buckets = round(len(poss_scalers) / self.state_reduction)

        if num_buckets < 1 or len(poss_scalers) == 0:
            return poss_scalers

        buckets = np.array_split(poss_scalers, num_buckets)
        final_scalers = []

        for bucket in buckets:
            if len(bucket) > 0:                    # Takes care if state_reduction is larger than starting buckets
                final_scalers.append(int(sum(bucket)/len(bucket)))

        return final_scalers

    else:
        return []
```

State Quality Function

Calculates resource, development and waste score separately

Each resource is further weighted with the loaded resource weights

Development score is weighted 10x to reflect priorities of an actively developing country

https://www.unep.org/news-and-stories/press-release/one-three-countries-world-lack-any-legally-mandated-standards

```python
def state_value(self) -> float:
    """Returns the base value of a state. Calculated
    as a weighted sum of the resources, developement and waste

    Returns:
        float: Base state value
    """

    resource_score =  (
        (self.weights['metalic_elm'] * self.metalic_elm) +
        (self.weights['timber'] * self.timber) +
        (self.weights['available_land'] * self.available_land) +
        (self.weights['water'] * self.water)
        )

    developement_score = (
        (self.weights['metalic_alloys'] * self.metalic_alloys) +
        (self.weights['electronics'] * self.electronics) +
        (self.weights['housing'] * self.housing) +
        (self.weights['farm'] * self.farm) +
        (self.weights['food'] * self.food)
        )

    waste_score = (
        (self.weights['metalic_waste'] * self.metalic_waste) +
        (self.weights['electronics_waste'] * self.electronics_waste) +
        (self.weights['housing_waste'] * self.housing_waste) +
        (self.weights['farm_waste'] * self.farm_waste) +
        (self.weights['food_waste'] * self.food_waste)
        )

    return round(resource_score + 10*developement_score - waste_score, 2)
```

EU Calculation

Captures base state value using previous function

Calculates probability of external collusion for every transfer in solution

Creates estimated discounted reward and the subsequent expected utility

Stores EU of solution at each step

https://buffett.northwestern.edu/news/2020/the-pitfalls-and-potential-of-international-cooperation.html

```python
def calculate_reward(self, solution: Solution):
    """Given the current solution, calculate the state
    quality, the undiscounted reward, the discounted reward,
    the probility a country accepts said transform (if applicable)
    The probility of success given self parameter c, and finally,
    the expected utility given the function defined in class.

    Args:
        solution (Solution): The current solution

    Returns:
        EU (float): Expected Utility of the state
    """

    C = 0.1
    new_state = solution.path[-1][1]
    curr_quality = new_state.state_value()
    og_quality = solution.path[0][1].state_value()

    other_country_probobility = []
    for step in solution.path:

        if type(step[0]) is Transfer:
            other_c_utility = self.countries[step[0].c_1_name].state_value()
            other_country_probobility.append(math.log(other_c_utility))

    if other_country_probobility:
        other_c_prob = sum(other_country_probobility) / len(other_country_probobility)

    else:
        other_c_prob = 1

    discounted_reward = round(pow(self.gamma, len(solution.path)+1) * (curr_quality - og_quality), 3)
    expected_utility = (other_c_prob * discounted_reward) + ((1 - other_c_prob) * C)

    solution.path[-1] += [expected_utility]
    solution.priority = expected_utility
```

Custom Implemented Priority Queue

Customly designed to utilize solution class

Priority is dictated by expected utility of current states in solution path

Utilizes heapq library for optimized binary tree sorting

https://towardsdatascience.com/introduction-to-python-heapq-module-53534feda625

```python
class PriorityQueue:

    queue: list
    max_size: int

    def __init__(self, maxsize: int):
        """Initilization function to set the maxsize
        of the priority queue

        Parameters:
            maxsize (int): Size of the queue
        """

        self.max_size = maxsize
        self.queue = []

    def push(self, item: Solution):
        """Adds new item into the queue utilizing heapq
        pop and push

        Parameters:
            item (Solution): New item to be added

        Returns:
            None
        """

        if len(self.queue) > self.max_size:
            heapq.heappop(self.queue)

        heapq.heappush(self.queue, item)

    def pop(self):
        """Pops the highest EU solution from the queue

        Returns:
            Solution: Highest EU solution in given queue
        """

        return heapq.heappop(self.queue)

    def empty(self):
        """Returns bool if the queue is empty or not

        Returns:
            bool: If the queue is empty
        """

        return len(self.queue) == 0
```

Other Considerations

Integrated max state reduction for deeper searches

Added resources: farm, food, available land, water, farm waste and food waste

Added subsequent transformations and relational values

Added relational dependencies between water to farm, water to food and farm to food

https://agriculture.vic.gov.au/farm-management/water/farm-water-solutions/how-much-water-does-my-farm-need#h2-10

https://www.ers.usda.gov/topics/farm-practices-management/irrigation-water-use/

https://www.waterfootprint.org/media/downloads/Report47-WaterFootprintCrops-Vol1.pdf

https://www.ers.usda.gov/amber-waves/2021/august/us-food-related-water-use-varies-by-food-category-supply-chain-stage-and-dietary-pattern/


```python
if self.state_reduction == -1:
    return [min(scalers)]
```

```python
@dataclass
class Country:

    name: str

    population: int
    metalic_elm: int
    timber: int
    available_land: int
    water: int

    state_reduction: int
    weights: ResourceWeights

    metalic_alloys: int = 0
    electronics: int = 0
    housing: int = 0
    farm: int = 0
    food: int = 0

    metalic_waste: int = 0
    electronics_waste: int = 0
    housing_waste: int = 0
    farm_waste: int = 0
    food_waste: int = 0
```

```python
@dataclass
class FoodTransform:

    scaler: int

    water_input: int
    farm_input: int

    food_output: int
    food_waste_output: int
    farm_output: int

    def __init__(self, scaler: int) -> None:
        """Given the state and the scaler, captures origional
        value and sets the new resource values of the state

        Parameters:
            state (Country): Current state to transform
            scaler (int): Scaler for transformations

        Returns:
            None
        """

        self.scaler = scaler

        self.water_input = 10 * scaler
        self.farm_input = 5 * scaler

        self.farm_output = 5 * scaler
        self.food_output = 1 * scaler
        self.food_waste_output = int(0.5 * scaler)
```

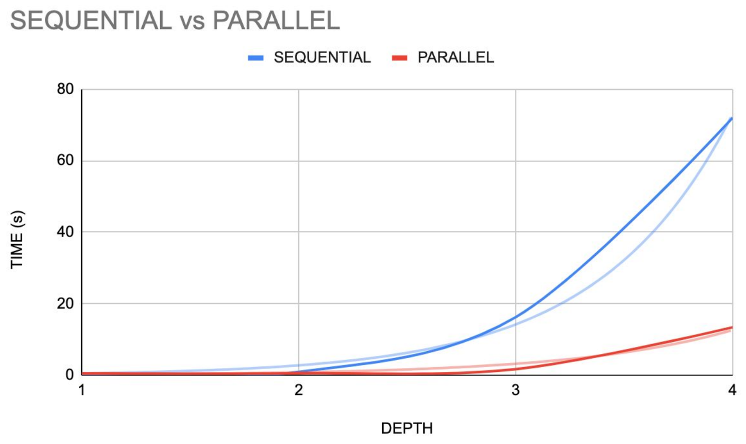
```python
@dataclass
class FarmTransform:

    scaler: int

    timber_input: int
    available_land_input: int
    water_input: int

    farm_output: int
    farm_waste_output: int

    def __init__(self, scaler: int) -> None:
        """Given the state and the scaler, captures origional
        value and sets the new resource values of the state

        Parameters:
            state (Country): Current state to transform
            scaler (int): Scaler for transformations

        Returns:
            None
        """

        self.scaler = scaler

        self.water_input = 10 * scaler
        self.timber_input = 5 * scaler
        self.available_land_input = 10 * scaler

        self.farm_output = 5 * scaler
        self.farm_waste_output = 1 * scaler
```

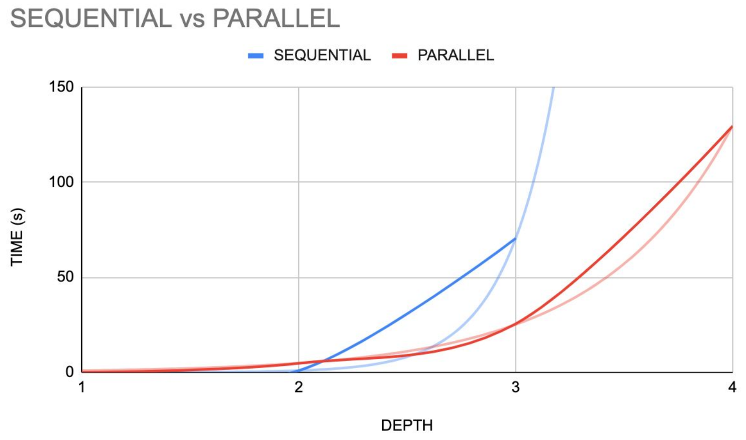
Sequential vs. Parallel Frontier Size

Exponential better parallel performance

Achieved through unique hybrid, depth-first search implementation

Reduction in searched states





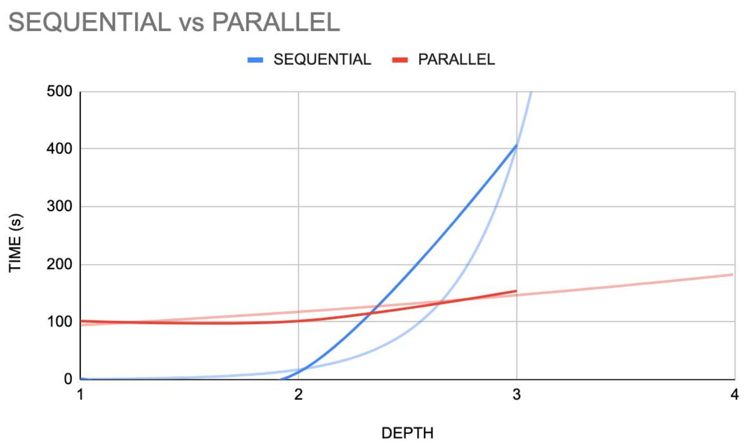
Citations: Citations: Citations

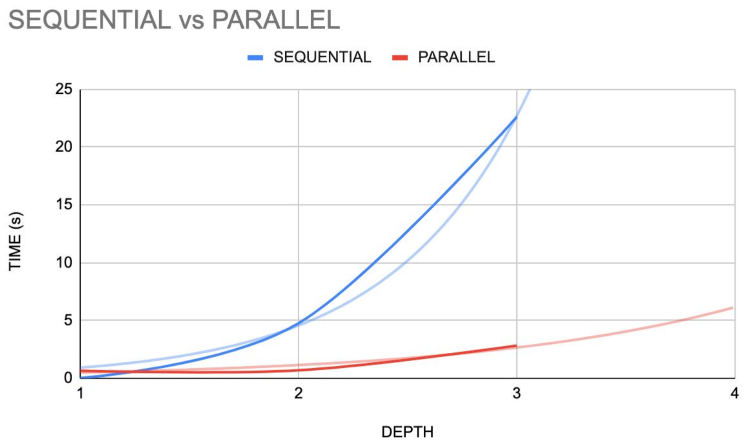
Sequential vs. Parallel State Reduction

Exponential better parallel performance

Achieved through unique hybrid, depth-first search implementation

Performance gain of parallel inversely correlated with state reduction





Base Level Run

Sets an initial ground truth to compare subsequent test cases off of

Heavy favor towards food and farm transforms

Utilizes Alloy transform

Base weights

Country	Population	Metalic_Elm	Timber	Available_Land	Water
Atlantis	100	700	2000	10000	5000
Brobdingnag	50	300	1200	6000	3000
Carpania	25	100	300	1500	750
Dinotopia	30	200	200	1000	500
Erewhon	70	500	1700	8500	4250

Resource	Weight
Population	0.1
Metalic_Elm	0.1
Timber	0.1
Available_Land	0.1
Water	0.2
Metlic_Alloys	0.4
Housing	0.5
Electronics	0.8
Farm	0.6
Food	0.8
Metalic_Waste	0.3
Electronic_Waste	0.3
Housing_Waste	0.2
Farm_Waste	0.1
Food_Waste	0.1

```
Expected Utility for This Action:
19702.354885765082
    FOOD TRANSFORM:
      INPUTS:
        water: 850
        farm: 425
      OUTPUTS:
        food: 85
        farm: 425
        food_waste: 42

Expected Utility for This Action: 125.44
    ALLOY TRANSFORM:
      INPUTS:
        population: 70
        metalic_elm: 140
      OUTPUTS:
        metalic_alloy: 70
        metalic_alloy_waste: 70
        population: 70

Expected Utility for This Action: 3776.922
    FARM TRANSFORM:
      INPUTS:
        water: 3400
        timber: 1700
        available_land: 3400
      OUTPUTS:
        farm: 1700
        farm_waste: 340

Expected Utility for This Action:
23347.348194490132
    TRANSFER:
      OTHER  -  SELF
      Atlantis -  Erewhon
      available_land  -  metalic_elm
      4  -  4
```

Decreased C - Value

Testing system with -500 c value

Causes catastrophic value state if transferred aren't successful

Model compensates via avoiding transfers, even if they would potentially be better

Country	Population	Metalic_Elm	Timber	Available_Land	Water
Atlantis	100	700	2000	10000	5000
Brobdingnag	50	300	1200	6000	3000
Carpania	25	100	300	1500	750
Dinotopia	30	200	200	1000	500
Erewhon	70	500	1700	8500	4250

Resource	Weight
Population	0.1
Metalic_Elm	0.1
Timber	0.1
Available_Land	0.1
Water	0.2
Metlic_Alloys	0.4
Housing	0.5
Electronics	0.8
Farm	0.6
Food	0.8
Metalic_Waste	0.3
Electronic_Waste	0.3
Housing_Waste	0.2
Farm_Waste	0.1
Food_Waste	0.1

```
Expected Utility for This Action:
23066.594278600976
    ALLOY TRANSFORM:
      INPUTS:
        population: 70
        metalic_elm: 140
      OUTPUTS:
        metalic_alloy: 70
        metalic_alloy_waste: 70
        population: 70

Expected Utility for This Action: 4595.712
    FARM TRANSFORM:
      INPUTS:
        water: 3400
        timber: 1700
        available_land: 3400
      OUTPUTS:
        farm: 1700
        farm_waste: 340
        food_waste: 42

Expected Utility for This Action:
31773.007171660793
    TRANSFER:
      OTHER  -  SELF
      Atlantis -  Erewhon
      timber -  metalic_elm
      5 -  5

Expected Utility for This Action:
27371.92091758638
    FOOD TRANSFORM:
      INPUTS:
        water: 850
        farm: 425
      OUTPUTS:
        food: 85
        farm: 425
```

Higher Waste Values

Waste values are exponentially higher

Simulates closer to restriction on 1st world countries

Reflecting reality, due to hight waste costs, model favors trading instead of producing domestically

Country	Population	Metalic_Elm	Timber	Available_Land	Water
Atlantis	100	700	2000	10000	5000
Brobdingnag	50	300	1200	6000	3000
Carpania	25	100	300	1500	750
Dinotopia	30	200	200	1000	500
Erewhon	70	500	1700	8500	4250

Resource	Weight
Population	0.1
Metalic_Elm	0.1
Timber	0.1
Available_Land	0.1
Water	0.2
Metlic_Alloys	0.4
Housing	0.5
Electronics	0.8
Farm	0.6
Food	0.2
Metalic_Waste	100
Electronic_Waste	400
Housing_Waste	300
Farm_Waste	300
Food_Waste	100

Expected Utility for This Action:
-5.039664093522874
TRANSFER:
OTHER - SELF
Dinotopia - Erewhon
metalic_elm - timber
2 - 2

Expected Utility for This Action:
-23.564223654418807
TRANSFER:
OTHER - SELF
Dinotopia - Erewhon
water - available_land
43 - 85

Expected Utility for This Action:
-19.744218324849438
TRANSFER:
OTHER - SELF
Dinotopia - Erewhon
water - available_land
43 - 85

Expected Utility for This Action:
-4.71247620242868
TRANSFER:
OTHER - SELF
Carpania - Erewhon
metalic_elm - timber
1 - 1