1 Barter

1.1 Problem Statement

The year is 2035 and AI robots have taken over the world. Consequently, the stock market has crashed, money is worth nothing, aliens should be arriving any minute, and humans have reverted back to hunter gatherer tribes- it's madness. Just like the movies!

Spring is just around the corner, the the weather is warming up, and you've decided that you want to plant some tomatoes. Unfortunately, you don't have any tomato seeds. Fortunately, you do have some old supplies you're looking to get rid of, and you can barter.

Here's what you have: A bell pepper, a wrench, three eggs, and a rope.

1. one bell pepper
2. one wrench
3. three eggs
4. one rope

Here's what others are willing to trade:

1. Jack will give you butter for two eggs
2. Emily will give you a chicken for a sheet of steel and a wrench
3. Rachel will give you nails for a butter
4. Sarah will give you paint for a pepper and rope
5. Mark will give you a sheet of steel for paint and an egg
6. Peter will give you tomato seeds for chicken and butter

1.2 Formalizing a Domain

1.2.1 Predicates

What predicates may be valuable here? Which ones can be general, and which need to be specific?

For example, Owns(person item) can be a general predicate. Maybe we want additional predicates for Person(x) and Item(y)? These might be redundant if we use additional predicates for each item type. For example: Egg(x), Paint(x), and so on. These predicates already encapsulate the assumptions that those

things are items.

Should other people in town be allowed to trade with one another, or only with us? How do we bake these into our problem definition. Work through the predicates and what they represent on the board.

1.2.2 Actions

What might a trade look like?
Transfer of ownership of course. Additionally, the people who are trading must have ownership to begin with. As before, can trades be allowed between other people? How many of each item does a person have? Can you retrade? Work through a few trades as actions, what are the arguments, preconditions, and effects?

1.2.3 PDDL Definition 1

In this sections is an example of a pddl file that finds a solution as well as the solution it finds.
Domain

```
(define (domain trade-strips)
(:predicates
(possess ?i)
(pepper ?i)
(wrench ?i)
(egg ?i)
(rope ?i)
(butter ?i)
(chicken ?i)
(steel ?i)
(seeds ?i)
(nails ?i)
(paint ?i))

(:action trade-jack
:parameters (?egg1 ?egg2 ?b)
:precondition (and  (possess ?egg1) (possess ?egg2)
(egg ?egg1) (egg ?egg2) (butter ?b))
:effect (and  (possess ?b) (not (possess ?egg1)) (not (possess ?egg2))))

(:action trade-emily
:parameters (?steel ?wrench ?chicken)
:precondition (and  (possess ?steel) (possess ?wrench)
(steel ?steel) (wrench ?wrench) (chicken ?chicken))
:effect (and  (possess ?chicken) (not (possess ?wrench))
```

```
(not (possess ?steel))))

(:action trade-rachel
:parameters (?nails ?butter)
:precondition (and  (possess ?butter) (nails ?nails) (butter ?butter))
:effect (and  (possess ?nails) (not (possess ?butter))))

(:action trade-sarah
:parameters (?paint ?pepper ?rope)
:precondition (and  (possess ?pepper) (possess ?rope)
(pepper ?pepper) (paint ?paint) (rope ?rope))
:effect (and  (possess ?paint)
(not (possess ?pepper)) (not (possess ?rope))))

(:action trade-mark
:parameters (?steel ?paint ?egg)
:precondition (and  (possess ?paint)
(possess ?egg) (paint ?paint) (egg ?egg) (steel ?steel))
:effect (and  (possess ?steel)
(not (possess ?egg)) (not (possess ?paint))))

(:action trade-peter
:parameters (?seeds ?chicken ?butter)
:precondition (and  (possess ?chicken)
(possess ?butter) (butter ?butter) (chicken ?chicken) (seeds ?seeds))
:effect (and  (possess ?seeds)
(not (possess ?chicken)) (not (possess ?butter)))))
```

Problem

```
(define (problem strips-trade1)
   (:domain trade-strips)
   (:objects pepper
             wrench
             egg1
             egg2
             egg3
             rope
             butter
             chicken
             steel
             seeds
             nails
             paint)
   (:init
    (possess pepper)
```

```
    (possess wrench)
    (possess egg1)
    (possess egg2)
    (possess egg3)
    (possess rope)
    (pepper pepper)
    (wrench wrench)
    (egg egg1)
    (egg egg2)
    (egg egg3)
    (rope rope)
    (butter butter)
    (chicken chicken)
    (steel steel)
    (seeds seeds)
    (nails nails)
    (paint paint))
   (:goal (and (possess seeds))))
```

Output

```
ff: found legal plan as follows

step    0: TRADE-JACK EGG1 EGG1 BUTTER
        1: TRADE-SARAH PAINT PEPPER ROPE
        2: TRADE-MARK STEEL PAINT EGG2
        3: TRADE-EMILY STEEL WRENCH CHICKEN
        4: TRADE-PETER SEEDS CHICKEN BUTTER
```

1.2.4 Discussion

1.2.5 Additional Points

What assumptions are made by this pddl definition? Ex: Can multiple trades occur? Why might they be problematic?

Examples

1. There is no concept of ownership for other towns members, only me. This could be problematic in tracking what other people own, and if they still have anything to trade.
2. If I make the same trade twice, I get duplicate items (redundant).
3. Not extremely scalable. Every trade is an action.

4. Can you find anymore?

How else could be represent the domain/problem statement, particularly to be more generic for future trade schemes.

1.2.6 PDDL Definition 2

In this sections is another example of a pddl file that finds a solution as well as the solution it finds.

Domain

```
(define (domain trade-strips)
(:predicates
(possess ?i)
(possesses ?p ?i)
(will-give ?p ?i1 ?i2)
(wants ?p ?i1 ?i2)
(is-of-type ?x1 ?x1-t))

(:action trade
:parameters (?person ?x1 ?x1-type ?x2 ?x2-type ?y1 ?y1-type ?y2 ?y2-type)
:precondition (and
(wants ?person ?x1-type ?x2-type)
(will-give ?person ?y1-type ?y2-type)
(is-of-type ?x1 ?x1-type)
(is-of-type ?x2 ?x2-type)
(is-of-type ?y1 ?y1-type)
(is-of-type ?y2 ?y2-type)
(possesses ?person ?y1)
(possesses ?person ?y2)
(possess ?x1)
(possess ?x2))
:effect (and
(not (possess ?x1))
(not (possess ?x2))
(possess ?y1)
(possess ?y2))
)

)
```

Problem

```
(define (problem strips-trade1)
   (:domain trade-strips)
   (:objects pepper
             wrench
```

```
            egg1
            egg2
            egg3
            rope
            butter
            chicken
            steel
            seeds
            nails
            paint
            null1
            null2
            null3
            null4
            pepper-type
            wrench-type
            egg-type
            rope-type
            butter-type
            chicken-type
            steel-type
            seeds-type
            nails-type
            paint-type
            null-type
            Jack
            Emily
            Rachel
            Sarah
            Mark
            Peter)
   (:init
    (possess pepper)
    (possess wrench)
    (possess egg1)
    (possess egg2)
    (possess egg3)
    (possess rope)
    (possess null3)
    (possess null4)
    (is-of-type pepper pepper-type)
    (is-of-type wrench wrench-type)
    (is-of-type egg1 egg-type)
    (is-of-type egg2 egg-type)
    (is-of-type egg3 egg-type)
    (is-of-type rope rope-type)
```

```
    (is-of-type butter butter-type)
    (is-of-type chicken chicken-type)
    (is-of-type steel steel-type)
    (is-of-type seeds seeds-type)
    (is-of-type nails nails-type)
    (is-of-type paint paint-type)
    (is-of-type null1 null-type)
    (is-of-type null2 null-type)
    (wants Jack egg-type egg-type)
    (wants Emily steel-type wrench-type)
    (wants Rachel butter-type null-type)
    (wants Sarah pepper-type rope-type)
    (wants Mark paint-type egg-type)
    (wants Peter chicken-type butter-type)
    (will-give Jack butter-type null-type)
    (will-give Emily chicken-type null-type)
    (will-give Rachel nails-type null-type)
    (will-give Sarah paint-type null-type)
    (will-give Mark steel-type null-type)
    (will-give Peter seeds-type null-type)
    (possesses Jack butter)
    (possesses Emily chicken)
    (possesses Rachel nails)
    (possesses Sarah paint)
    (possesses Mark steel)
    (possesses Peter seeds)
    (possesses Jack null1)
    (possesses Emily null1)
    (possesses Rachel null1)
    (possesses Sarah null1)
    (possesses Mark null1)
    (possesses Peter null1)
    (possesses Jack null2)
    (possesses Emily null2)
    (possesses Rachel null2)
    (possesses Sarah null2)
    (possesses Mark null2)
    (possesses Peter null2)
    )
   (:goal (and (possess seeds))))
```

Output

```
step    0: TRADE JACK EGG1 EGG-TYPE EGG1 EGG-TYPE BUTTER BUTTER-TYPE NULL1 NULL-TYPE
        1: TRADE SARAH PEPPER PEPPER-TYPE ROPE ROPE-TYPE PAINT PAINT-TYPE NULL1 NULL-TYPE
        2: TRADE MARK PAINT PAINT-TYPE EGG2 EGG-TYPE STEEL STEEL-TYPE NULL1 NULL-TYPE
        3: TRADE EMILY STEEL STEEL-TYPE WRENCH WRENCH-TYPE CHICKEN CHICKEN-TYPE NULL1 NULL-TYPE
        4: TRADE PETER CHICKEN CHICKEN-TYPE BUTTER BUTTER-TYPE SEEDS SEEDS-TYPE NULL1 NULL-TYPE
```

1.3 Discussion

What may be some benefits/concerns with this representation? What previous concerns may it address?

The second solution is more general than the previous. It puts the business logic into the problem definition, rather than the domain. This allows for more problems to be developed and solved using the same domain.

What are some downsides? The main one is the requirement of using null types. It is not as readable, and doesn't really make sense to trade nulls. Additionally, the trade action is always going to need to support the largest possible number of inputs and outputs for the trade. You could write additional actions for every permutation of input and output numbers, and the trades could be tailored for each. For example one trade takes one in and outputs one, one takes in one and outputs two, and so on. This removes the need for nulls, but requires more domain code.

This is a fairly open ended problem. **Can you think of any others representations? What are the trade offs?**