CS/ECE/ISyE 524 — Introduction to Optimization — Summer 2021

Optimizing Catan

Ayden Schultz (amschultz7@wisc.edu), Jason Bai (jwbai@wisc.edu)

Table of Contents

1. Introduction

Since its inception in 1995, Settlers of Catan has been a game known for its “easy to learn, hard to master” gameplay, being a relatively simple game on the surface with a high strategic skill ceiling. There are several annual tournaments for the game, and since its release has sold over 30 million copies in over 40 different languages. Therefore, due to the game’s immense popularity and high skill gameplay, we thought this would be a good game to try and optimize. Because the turn-to-turn gameplay of Catan has so many moving parts, we knew it would be best to hone in on a specific, but vital, mechanic, that being settlement placement.

In Settlers of Catan, up to 4 players compete to colonize an island called Catan. Each player advances their position by building settlements and cities, with roads connecting them, across the game board. To create these various structures, players need to spend combinations of resources. There are 5 resources, including: Clay, Wood, Sheep, Wheat, and Ore. These resources can be attained by players by harvesting

them from the game board. The game board itself is made up of hexagonal tiles, each representing a resource to be harvested.

Each of these tiles has a number laid on top of it as well that is in the range of 2-12 (the set of numbers attainable from rolling 2 six-sided dice). From turn to turn, players roll 2 six-sided dice, and if the number rolled matches one that sits on a resource, that resource can be harvested. Players can then exchange these resources to build settlements, roads, cities, and developmental cards. Players would then place a settlement on any open corner, and cities would replace already existing settlements. Roads would be placed along edges and these would connect settlements and cities. So settlements are important in the early of the game, roads are important all throughout the game, and cities are only useful at the end of the game. Development cards are extremely powerful cards with extremely powerful effects. One such example of a card is that it allows a player to take all the wood from every player.


Road
= 0 Victory Points
Settlement
= 1 VP
City
= 2 VPs
Development
Card
= ? VPs

Only players with settlements or cities on the corner of the resource tile will harvest the resource when it is rolled. For example, if a player had a settlement on the circled corner, and that player rolls a 5, then that player would get wheat. Certain numbers are more likely to be rolled, as different numbers can be attained from different sums of the two dice. For example, a tile with the number 12 is less valuable than a tile with the number eight, as 12 can only be attained from rolling two sixes, and 8 can be attained from rolling two fours, a three and five, a five and three, etc. If a player rolls a 7, then they get to move the robber. Onto a different hexagon. If a robber were to be placed on the hexagon with the red 8 at the top, then if a player rolled an 8, then they would not get resources. There are also harbors, which are labeled by the ships on the game board. Having a settlement with access to a harbor lets the player trade in 2 of the resource on the sail of the ship for any one other resource. Certain harbors offer a trade of any 3 resources for any one resource. These harbors, due to their flexibility, are considered more valuable than other harbors in our model.


12
9
6
10
2:1

The purpose of our model is then, given a hypothetical game board, where tiles and numbers are randomized, and some possible placement spots may be taken, decide where a set of 3 settlements could be placement to maximize resource gain. We have chosen to find 3 positions, as in the first turn of the game, each player places 2 settlements anywhere on the game board at no cost, and we wanted to provide a third back up option. Settlements cannot be placed next to each other. To expand the reach of our model past the first turn, we have chosen to include a time value from 1-3. These values represent 1: early game, 2: mid-game, and 3: late-game. Certain resources are more valuable are different stages of the game, as different structures require different resources, and certain structures become more or less useful as the game goes on. Because of this, we have weighted the value of resource types depending on the stage of the game. For example, the brick resource is very useful in the early game, as it can be used to build roads and settlements. As such, it is more heavily weighted in the early game than the late game, where it is less useful. We are assuming with our model that players can reach the position found optimal by the model. Since settlements must be connected by road after the first two are placed, there are certain scenarios where an optimal placement spot may be unreachable. We also will not be taking cities into account, nor will we be "exchanging" resources to build roads. In our model, we will not be taking the robber into account. Additionally, the way we implement harbors is different from the game. This will be further explained in the mathematical model. Finally, we will not be taking the effects of the development cards into account.

Four Variations on the Model

The main variation is what we believe is the best distribution of resources at each stage of the game. Wood is important for all 3 stages of the game. Sheep are only used for settlements, so they are only important at the early of the game. Bricks are important in all 3 stages of the game because they are used to make roads and settlements. Ore is only used to make cities, so it is only important at the end of the game. For this variation, we will be using two different data sets. The second variation is a strategy where we manipulated the model to use less general strategies. For our third solution, we choose to search for the corners that would optimize the production of resources needed to produce roads. We only want wood and bricks at all stages of the game, as wood and bricks are used to make roads. For this variation, we only used 1 set of data. The final variation on the model to choose tiles that produce the resources required for development cards (ore, sheep, and wheat). For this variation, we only used 1 set of data.

2. Mathematical model

This optimization problem is a MIP. This is an MIP problem because the goal of this problem is to find which 3 corners we should put settlements on that would yield the greatest total resource. In order to do this, we have a variable x for each corner and it indicates whether there should be a settlement on that corner or not. Meaning that each x has to be 0 or 1. Since there is at least one variable that requires an integer value, that would make this an integer problem. Additionally, In order to execute this, one of our constraints is quadratic. Thus making this problem a MIP.

2.1 Main Variation

Data:

1. $resources$ = The possible resources obtainable.
   - :nothing is used in two ways.
     - First is that if a corner does not touch 3 hexagons. Second is that when we calculate the amount of additional resources that harbors bring in, we associate the amount of resources that we give up in this :nothing column. The reason for this is that it makes coding it a lot easier and reduces the number of lines of code that we use

1. $time$ = The 3 stages of the game
   - 1, 2, 3 represents beginning, middle and end respectively

1. $resourceValues_{ir}$ = The value of resource r at time t.

1. $resourcePerCorner_i$ = The possible resources that can be obtained from each corner i if a settlement is present on that corner.
   - Each corner can have 3 possible resources.
   - If there is not 3, then that corner will have ":nothing" as its resource.

1. $chancesPerCorner_i$ = The chance of rolling each resource that is present at each corner i. This corresponds to $resourcePerCorner_i$.
   - For Example:
     - $chancesPerCorner_1$ = $[0, 0, 2]$
     - $resourcePerCorner_1$ = $[:nothing, :nothing, :wood]$
     - The chance of obtaining wood is 2.

1. $edges$ = Each edge on the game board.

1. $harbor$ = The harbors in the game
   - We have the standard harbors for each resource.
   - The :anything harbor represents harbors where you can get any resource you want from them.

1. $harborValues_h$ = The value of each harbor h

1. $harborCorners_{ch}$ = Shows which harbor has corners. Corner c has harbor h.

1. $resourcesTime1_{ir}$ = The amount of resources r at each corner i at time 1 if a settlement is present on that corner.

- The amount of resources affected by harbors is taken into account.

1. $resourcesTime2_{ir}$ = The amount of resources r at each corner i at time 2 if a settlement is present on that corner.
   - The amount of resources affected by harbors is taken into account.

1. $resourcesTime3_{ir}$ = The amount of resources r at each corner i at time 3 if a settlement is present on that corner.
   - The amount of resources affected by harbors is taken into account.

1. $totalResources_i$ = The total amount of resources at each corner i at all times combined if a settlement is present on that corner. This is obtained by adding $resourcesTime1_{ir}$, $resourcesTime2_{ir}$, and $resourcesTime3_{ir}$ for each corner i and resource r.

Calculating Harbors:

- Calculating the resource gain for all the harbors except for :anything harbors:
  - The harbor value associated with the resource is added to the resource at the correct corner.
    - For Example:
      - If corner 9 had a wood harbor, then it would add the harbor value of wood into the wood total on corner 9
      - This is done for each time.

- Calculating the resource gain for the :anything harbor
  - If a corner has an :anything harbor, then each resource on that corner is increased by a fifth of the value of the :anything harbor
    - This evenly distributes the value of the :anything harbor onto each resource because any resource is obtainable.

- Exchanging Process
  - First we calculate the amount of total resources that is produced by the corner with the harbor
  - Then we take 30% of each total resource at each corner, and make it negative
    - We store this variable in the :nothing column
      - As stated before, the reason for this is that it makes coding it a lot easier and reduces the number of lines of code that we use.
      - When we calculate $totalResources_{ir}$, then this will be subtracted properly.

- Why calculate resource gain and exchange from harbors like this?
  - As you can see, this is not the same way that harbors work in the game. As stated in the introduction, one would exchange 2 or 3 of any resource to obtain 1 of the resource that the harbor offers. The ratio of losing to gaining resource is (2 or 3) to 1. In our case, almost all of the time, if not all of the time, the amount of resources that we gain is significantly more than the amount of resources that we give up. This is in reverse of the actual game.

- The reason why we do this is because we believe that to have the option to exchange resources is very significant. This is very powerful as you can essentially pick and choose when you want a certain resource and when you do not.
- For example: In one scenario you might have too many bricks and not enough sheep, so you exchange the excess bricks for sheep. In another scenario you could have the same resource of having too many bricks and barely any sheep, but in this scenario you need to make a lot of roads. So you decide not to exchange bricks for sheep. Roads require bricks and not sheep. To have this option to exchange means that in many scenarios you would not have to wait until you roll for resources that you need. Which would speed up the time it takes for you to get more settlements, roads, and cities.
- We want this powerful option to be reflected when we calculate which corner is optimal or not. Everytime when we played the game, we realized that having a corner with a harbor is extremely powerful and we would always try to get one ASAP, despite the resource yield being low. Which is why we decided that any corner with a harbor will result in a gain of resources instead of breaking even.

Decision Variables:

1. x_i = The x variable essentially acts like a switch that says whether or not a given corner i has a settlement.
   - If x is 0 then that corner has a settlement.
   - If x is 1 then that corner does not have a settlement.

Constraints:

Constraints on x_i:

1. Makes sure that x will only be 0 or 1. The only numbers whose square is itself are 0 and 1.

$$x_i^2 - x_i <= 0, \forall i \in corners$$

1. Makes sure that only 3 corners will be picked to have a settlement on them.

$$\sum_{i \in corners} x_i <= 3 - \sum_{i \in corners} x_i <= -3$$

1. At least one of the corners that we put a settlement on must be a harbor. :nothing will always be a negative number and :nothing will always be less than -0.01.

$$\sum_{i \in corners} x_i \times resourceTime1_{i,nothing} + x_i \times resourceTime2_{i,nothing} + x_i \times resourceTime3_{i,nothing} <= -0.01$$

Material Constraints:

The material constraints make sure that we have enough of each resource at the necessary time.

1. We must have at least 36 Wood in all 3 stages of the game because Wood is used for roads and settlements. Roads are used throughout the entire game, while settlements are more important in the early game.

$$-\sum_{i \in corners} x_i \times resourceTime1_{i,wood} + x_i \times resourceTime2_{i,wood} + x_i \times resourceTime3_{i,wood} <= -36$$

1. We must have at least 20 sheep in the beginning of the game because sheep are only used for settlements. Sheep are used for settlements only, and settlements are most important in the early game.

$$-\sum_{i \in corners} x_i \times resourceTime1_{i,sheep} <= -20$$

1. We must have at least 100 bricks in all 3 stages of the game because bricks are used for roads and settlements. Roads are used throughout the entire game, while settlements are more important in the early game.

$$-\sum_{i \in corners} x_i \times resourceTime1_{i,bricks} + x_i \times resourceTime2_{i,bricks} + x_i \times resourceTime3_{i,bricks} <= -100$$

1. We must have at least 25 ore at the end of the game because ore is only used to make cities. Cities are only usable if we already have settlements, so thus ore is only useful at the end of the game.

$$-\sum_{i \in corners} x_i \times resourceTime3_{i,ore} <= -25$$

1. We must have at least 47 wheat at the middle and end of the game because wheat is important for settlements and cities. Although settlements are important for the early game, they are nowhere near as important as roads and bricks. The main importance of wheat is for making more settlements, which will be more prevalent at the middle of the game, and making cities, which are important at the end of the game.

$$-\sum_{i \in corners} x_i \times resourceTime2_{i,wheat} + x_i \times resourceTime3_{i,wheat} <= -47$$

Distance Constraint:

1. Two settlements cannot be next to each other. That means for each edge, only one of the corners can have a settlement. Since x is either 1 or 0, this constraint makes sure that only one of the x's on each edge can be 1.

$$x_i + x_j <= 1, \forall (i,j) \in edges$$

Objective:

1. The objective is to find which corners will give us the greatest total resource. This is the sum of the total resources at each corner i multiplied by x_i. Since x_i is either 0 or 1, we are going to get a solution where 3 of the corners are going to be 1, and 51 of the corners are going to be 0. This objective function will basically figure out which 3 x_i variables (corners that we put settlements on) should be 1 that will result in the greatest total resource.

$$\sum_{i \in corners} x_i \times totalResources_i$$

2.2 Road Variation

The only difference from the main variation is the material constraints.

Material Constraints:

The material constraints make sure that we have enough of each resource at the necessary time.

1. We must have at least 36 wood in all 3 stages of the game because we are prioritizing making roads, and wood is one of the ingredients to make roads.

$$-\sum_{i \in corners} x_i \times resourceTime1_{i,wood} + x_i \times resourceTime2_{i,wood} + x_i \times resourceTime3_{i,wood} <= -36$$

1. We must have at least 100 bricks in all 3 stages of the game because in this variation, we are prioritizing making roads, and bricks are one of the ingredients to make roads.

$$-\sum_{i \in corners} x_i \times resourceTime1_{i,bricks} + x_i \times resourceTime2_{i,bricks} + x_i \times resourceTime3_{i,bricks} <= -100$$

2.3 Development Cards Variation

The only difference from the main variation is the material constraints.

Material Constraints:

The material constraints make sure that we have enough of each resource at the necessary time.

1. We must have at least 40 sheep in all 3 stages of the game because we are prioritizing making development cards, and sheep is one of the resources to get development card.

$$-\sum_{i \in corners} x_i \times resourceTime1_{i,sheep} + x_i \times resourceTime2_{i,sheep} + x_i \times resourceTime3_{i,sheep} <= -40$$

1. We must have at least 50 ore in all 3 stages of the game because we are prioritizing making development cards, and ore is one of the resources to get development cards.

$$-\sum_{i \in corners} x_i \times resourceTime1_{i,ore} + x_i \times resourceTime2_{i,ore} + x_i \times resourceTime3_{i,ore} <= -50$$

1. We must have at least 80 wheat in all 3 stages of the game because we are prioritizing making development cards, and wheat is one of the ingredients to resources to get development cards.

$$-\sum_{i \in corners} x_i \times resourceTime1_{i,wheat} + x_i \times resourceTime2_{i,wheat} + x_i \times resourceTime3_{i,wheat} <= -80$$

3. Solution

Main Variation Dataset 1

In [4]:

```
using JuMP, Clp, NamedArrays, Gurobi

#The possible resources
resources = [:wood, :wheat, :brick, :sheep, :ore, :nothing]

#Each time
time = [1, 2, 3]

#The values for resources over each time
resource_values = NamedArray([8 4 8 4 1 0; 5 7 5 2 3 0; 3 8 3 2 8 0], (time, resources), ("Time", "Resources"))
```

```
#Every possible corner
corners = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,

#The possible resources on each corner
resource_per_corner = NamedArray([:nothing :nothing :wood; :nothing :nothing :ore; :nothing :nothing :sheep; :nothing :nc

#The chance of getting a resource on each corner
chances_per_corner = NamedArray([0 0 2; 0 0 3; 0 0 1; 0 0 2; 0 2 3; 0 3 1; 0 1 0; 0 2 5; 2 3 3; 3 1 5; 1 0 5; 0 0 5; 2 5

#The edges connecting each corner
edges = [[1,4], [1,5], [2,5], [2,6], [6,3], [3,7], [4,8], [5,9], [6, 10], [7,11], [8,12], [8,13], [9, 13], [9,14], [10,14

#Every possible harbor
harbor = [:anything, :brick, :wood, :sheep, :ore, :wheat]

#The values for each harbor
harbor_values = Dict(zip(harbor, [7, 4, 4, 4, 4, 4]))

#The corners that have harbors
harbor_corners = [[2, :anything], [5, :anything], [3, :anything], [7, :anything],[16, :brick], [21, :brick],[27, :anythir

#Initializing each array
list1 = zeros(54,6)
list2 = zeros(54,6)
list3 = zeros(54,6)

#The number of resources for time 1
resources_time_1 = NamedArray(list1, (corners, resources), ("Corners", "Amount of Resources"))

#The number of resources for time 1
resources_time_2 = NamedArray(list2, (corners, resources), ("Corners", "Amount of Resources"))

#The number of resources for time 1
resources_time_3 = NamedArray(list3, (corners, resources), ("Corners", "Amount of Resources"))

#Finds the amount of each resource for each time. Not including harbors
for i in 1:54
    for j in 1:3
        resources_time_1[i, resource_per_corner[i, j]] = resource_values[1, resource_per_corner[i, j]]*chances_per_corner
        resources_time_2[i, resource_per_corner[i, j]] = resource_values[2, resource_per_corner[i, j]]*chances_per_corner
        resources_time_3[i, resource_per_corner[i, j]] = resource_values[3, resource_per_corner[i, j]]*chances_per_corner
    end
end

#Finds the amount of additional resources that each harbor brings
for h in harbor_corners
```

```
    #This is the amount of resources that we "trade in" for each harbor
    total_resource_to_be_subtracted_1 = 0
    total_resource_to_be_subtracted_2 = 0
    total_resource_to_be_subtracted_3 = 0
    for j in resources
        total_resource_to_be_subtracted_1 += resources_time_1[h[1], j]
        total_resource_to_be_subtracted_2 += resources_time_2[h[1], j]
        total_resource_to_be_subtracted_3 += resources_time_3[h[1], j]
    end

    #Adding the amount of of additional resources that the harbor brings in
    #Does not include :anything
    if h[2] != :anything
        resources_time_1[h[1], h[2]] += harbor_values[h[2]]
        resources_time_2[h[1], h[2]] += harbor_values[h[2]]
        resources_time_3[h[1], h[2]] += harbor_values[h[2]]
    end

    #Adding the amount of of additional resources that the harbor brings in
    #Includes :anything
    if h[2] == :anything
        for r in resources
            resources_time_1[h[1], r] += harbor_values[h[2]]/5
            resources_time_2[h[1], r] += harbor_values[h[2]]/5
            resources_time_3[h[1], r] += harbor_values[h[2]]/5
        end
    end

    #The amount of total resources that we lose for the harbor
    #This value is stored in the row of :nothing for convinience
    resources_time_1[h[1], :nothing] = -(total_resource_to_be_subtracted_1*0.3)
    resources_time_2[h[1], :nothing] = -(total_resource_to_be_subtracted_2*0.3)
    resources_time_3[h[1], :nothing] = -(total_resource_to_be_subtracted_3*0.3)
end

#The total resources over all the times
total_resources = zeros(54)
for i in 1:54
    for j in resources
        total_resources[i] += resources_time_1[i, j]
        total_resources[i] += resources_time_2[i, j]
        total_resources[i] += resources_time_3[i, j]
    end
end
```

In [5]:
```
m = Model(Gurobi.Optimizer)

#Decision Variable x for each corner
@variable(m, x[1:54])
;
```

```
Academic license - for non-commercial use only
```

In [6]:
```
#Constraint on x[1:54]

#x has to be 0 or 1
@constraint(m, x_one_zero[i in 1:54], x[i]^2 == x[i])

#The total number of corners chosen is 3
@constraint(m, x_choose_three, sum(x[i] for i in 1:54) == 3)

#There has to be at least 1 harbor
@constraint(m, x_at_least_one_harbor, sum(x[i]*resources_time_1[i, :nothing]+x[i]*resources_time_3[i, :nothing]+x[i]*resc
;
```

In [7]:
```
#Material Constraints

#Constraint for amount of wood
@constraint(m, wood, sum(resources_time_1[i, :wood]*x[i] + resources_time_2[i, :wood]*x[i] + resources_time_3[i, :wood]*x

#Constraint for amount of sheep
@constraint(m, sheep, sum(resources_time_1[i, :sheep]*x[i] for i in 1:54)>= 20)

#Constraint for amount of brick
@constraint(m, brick, sum(resources_time_1[i, :brick]*x[i] + resources_time_2[i, :brick]*x[i] + resources_time_3[i, :bric

#Constraint for amount of ore
@constraint(m, ore, sum(resources_time_3[i, :ore]*x[i] for i in 1:54)>= 25)

#Constraint for amount of wheat
@constraint(m, wheat, sum(resources_time_2[i, :wheat]*x[i] for i in 1:54) + sum(resources_time_3[i, :wheat]*x[i] for i in
;
```

In [8]:
```
#Distance Constraints
for i in edges
```

```
    @constraint(m, x[i[1]]+ x[i[2]] <= 1)
end
```

In [9]:

```
#Objective Function
@objective(m, Max, sum(x[i]*total_resources[i] for i in 1:54))
;
```

In [10]:

```
set_optimizer_attribute(m, "NonConvex", 2)
optimize!(m)
```

```
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 79 rows, 54 columns and 360 nonzeros
Model fingerprint: 0xe280017d
Model has 54 quadratic constraints
Coefficient statistics:
  Matrix range     [1e+00, 8e+01]
  QMatrix range    [1e+00, 1e+00]
  QLMatrix range   [1e+00, 1e+00]
  Objective range  [2e+01, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e-02, 1e+02]

Continuous model is non-convex -- solving as a MIP.

Presolve time: 0.01s
Presolved: 241 rows, 54 columns, 468 nonzeros
Presolved model has 54 bilinear constraint(s)
Variable types: 54 continuous, 0 integer (0 binary)

Root relaxation: objective 5.270822e+02, 73 iterations, 0.01 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  527.08225    0    5          -  527.08225      -     -    0s
     0     0  526.48225    0    5          -  526.48225      -     -    0s
     0     0  526.48225    0    5          -  526.48225      -     -    0s
     0     0  526.27886    0   10          -  526.27886      -     -    0s
     0     0  526.27886    0   10          -  526.27886      -     -    0s
     0     2  526.27886    0   10          -  526.27886      -     -    0s
*   54    22              15     341.6000000  510.47919  49.4%   3.4    0s
*  106    38              25     356.8000000  510.47919  43.1%   3.8    0s
*  116    46              13     384.0000000  510.47919  32.9%   4.0    0s
*  149    32              17     388.9000000  509.23086  30.9%   3.9    0s
```

```
*  221    40              16     404.1000000  484.43586  19.9%   3.2    0s

Cutting planes:
  RLT: 3

Explored 367 nodes (1107 simplex iterations) in 0.24 seconds
Thread count was 4 (of 4 available processors)

Solution count 5: 404.1 388.9 384 ... 341.6

Optimal solution found (tolerance 1.00e-04)
Best objective 4.041000000000e+02, best bound 4.041000000000e+02, gap 0.0000%
```

In [11]:

```
println("Solution")
println("The Best Corners To Place Are:")
for i in 1:54
    if value(x[i])==1
        println("   Corner ", i, ": ", value(x[i]))
    end
end
```

```
Solution
The Best Corners To Place Are:
   Corner 13: 1.0
   Corner 17: 1.0
   Corner 42: 1.0
```

Main Variation Dataset 2

In [12]:

```
using JuMP, Clp, NamedArrays, Gurobi

#The possible resources
resources = [:wood, :wheat, :brick, :sheep, :ore, :nothing]

#Each time
time = [1, 2, 3]

#The values for resources over each time
resource_values = NamedArray([8 4 8 4 1 0; 5 7 5 2 3 0; 3 8 3 2 8 0], (time, resources), ("Time", "Resources"))

#Every possible corner
corners = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,

#The possible resources on each corner
resource_per_corner = NamedArray([:nothing :nothing :sheep; :nothing :nothing :sheep; :nothing :nothing :wheat; :nothing
```

```
#The chance of getting a resource on each corner
chances_per_corner = NamedArray([0 0 5; 0 0 3; 0 0 2; 0 0 5; 0 5 3; 0 3 2; 0 2 0; 0 5 3; 5 3 2; 3 2 0; 2 0 1; 0 0 3; 5 3

#The edges connecting each corner
edges = [[1,4], [1,5], [2,5], [2,6], [6,3], [3,7], [4,8], [5,9], [6, 10], [7,11], [8,12], [8,13], [9, 13], [9,14], [10,14

#Every possible harbor
harbor = [:anything, :brick, :wood, :sheep, :ore, :wheat]

#The values for each harbor
harbor_values = Dict(zip(harbor, [7, 4, 4, 4, 4, 4]))

#The corners that have harbors
harbor_corners = [[2, :anything], [5, :anything], [3, :anything], [7, :anything], [8, :sheep], [12, :sheep], [16, :brick]

#Initializing each array
list1 = zeros(54,6)
list2 = zeros(54,6)
list3 = zeros(54,6)

#The number of resources for time 1
resources_time_1 = NamedArray(list1, (corners, resources), ("Corners", "Amount of Resources"))

#The number of resources for time 1
resources_time_2 = NamedArray(list2, (corners, resources), ("Corners", "Amount of Resources"))

#The number of resources for time 1
resources_time_3 = NamedArray(list3, (corners, resources), ("Corners", "Amount of Resources"))

#Finds the amount of each resource for each time. Not including harbors
for i in 1:54
    for j in 1:3
        resources_time_1[i, resource_per_corner[i, j]] = resource_values[1, resource_per_corner[i, j]]*chances_per_corner
        resources_time_2[i, resource_per_corner[i, j]] = resource_values[2, resource_per_corner[i, j]]*chances_per_corner
        resources_time_3[i, resource_per_corner[i, j]] = resource_values[3, resource_per_corner[i, j]]*chances_per_corner
    end
end

#Finds the amount of additional resources that each harbor brings
for h in harbor_corners

    #This is the amount of resources that we "trade in" for each harbor
    total_resource_to_be_subtracted_1 = 0
    total_resource_to_be_subtracted_2 = 0
    total_resource_to_be_subtracted_3 = 0
```

```
    for j in resources
        total_resource_to_be_subtracted_1 += resources_time_1[h[1], j]
        total_resource_to_be_subtracted_2 += resources_time_2[h[1], j]
        total_resource_to_be_subtracted_3 += resources_time_3[h[1], j]
    end

    #Adding the amount of of additional resources that the harbor brings in
    #Does not include :anything
    if h[2] != :anything
        resources_time_1[h[1], h[2]] += harbor_values[h[2]]
        resources_time_2[h[1], h[2]] += harbor_values[h[2]]
        resources_time_3[h[1], h[2]] += harbor_values[h[2]]
    end

    #Adding the amount of of additional resources that the harbor brings in
    #Includes :anything
    if h[2] == :anything
        for r in resources
            resources_time_1[h[1], r] += harbor_values[h[2]]/5
            resources_time_2[h[1], r] += harbor_values[h[2]]/5
            resources_time_3[h[1], r] += harbor_values[h[2]]/5
        end
    end

    #The amount of total resources that we lose for the harbor
    #This value is stored in the row of :nothing for convinience
    resources_time_1[h[1], :nothing] = -(total_resource_to_be_subtracted_1*0.3)
    resources_time_2[h[1], :nothing] = -(total_resource_to_be_subtracted_2*0.3)
    resources_time_3[h[1], :nothing] = -(total_resource_to_be_subtracted_3*0.3)
end

#The total resources over all the times
total_resources = zeros(54)
for i in 1:54
    for j in resources
        total_resources[i] += resources_time_1[i, j]
        total_resources[i] += resources_time_2[i, j]
        total_resources[i] += resources_time_3[i, j]
    end
end

m = Model(Gurobi.Optimizer)

#Decision Variable x for each corner
@variable(m, x[1:54])
;
```

```
#Constraint on x[1:54]

#x has to be 0 or 1
@constraint(m, x_one_zero[i in 1:54], x[i]^2 == x[i])

#The total number of corners chosen is 3
@constraint(m, x_choose_three, sum(x[i] for i in 1:54) == 3)

#There has to be at least 1 harbor
@constraint(m, x_at_least_one_harbor, sum(x[i]*resources_time_1[i, :nothing]+x[i]*resources_time_3[i, :nothing]+x[i]*resc
;

#Material Constraints

#Constraint for amount of wood
@constraint(m, wood, sum(resources_time_1[i, :wood]*x[i] + resources_time_2[i, :wood]*x[i] + resources_time_3[i, :wood]*x

#Constraint for amount of sheep
@constraint(m, sheep, sum(resources_time_1[i, :sheep]*x[i] for i in 1:54)>= 20)

#Constraint for amount of brick
@constraint(m, brick, sum(resources_time_1[i, :brick]*x[i] + resources_time_2[i, :brick]*x[i] + resources_time_3[i, :bric

#Constraint for amount of ore
@constraint(m, ore, sum(resources_time_3[i, :ore]*x[i] for i in 1:54)>= 25)

#Constraint for amount of wheat
@constraint(m, wheat, sum(resources_time_2[i, :wheat]*x[i] for i in 1:54) + sum(resources_time_3[i, :wheat]*x[i] for i in
;

#Distance Constraints
for i in edges
    @constraint(m, x[i[1]]+ x[i[2]] <= 1)
end


#Objective Function
@objective(m, Max, sum(x[i]*total_resources[i] for i in 1:54))
;

set_optimizer_attribute(m, "NonConvex", 2)
optimize!(m)


println("Solution")
```

```
println("The Best Corners To Place Are:")
for i in 1:54
    if value(x[i])==1
        println("   Corner ", i, ": ", value(x[i]))
    end
end
```

```
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 79 rows, 54 columns and 335 nonzeros
Model fingerprint: 0x1c055e39
Model has 54 quadratic constraints
Coefficient statistics:
  Matrix range     [1e+00, 9e+01]
  QMatrix range    [1e+00, 1e+00]
  QLMatrix range   [1e+00, 1e+00]
  Objective range  [1e+01, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e-02, 1e+02]

Continuous model is non-convex -- solving as a MIP.

Presolve time: 0.00s
Presolved: 241 rows, 54 columns, 443 nonzeros
Presolved model has 54 bilinear constraint(s)
Variable types: 54 continuous, 0 integer (0 binary)

Root relaxation: objective 4.627500e+02, 46 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  462.75000    0    3          -  462.75000      -     -    0s
     0     0  457.48368    0    5          -  457.48368      -     -    0s
     0     0  457.48368    0    4          -  457.48368      -     -    0s
     0     0  457.32743    0    7          -  457.32743      -     -    0s
     0     0  457.01493    0    7          -  457.01493      -     -    0s
     0     2  457.01493    0    7          -  457.01493      -     -    0s
*   38     0               6     328.6000000  328.60000  0.00%   2.3    0s

Cutting planes:
  RLT: 3

Explored 39 nodes (139 simplex iterations) in 0.05 seconds
Thread count was 4 (of 4 available processors)

Solution count 1: 328.6
```

```
Optimal solution found (tolerance 1.00e-04)
Best objective 3.286000000000e+02, best bound 3.286000000000e+02, gap 0.0000%
Solution
The Best Corners To Place Are:
   Corner 8: 1.0
   Corner 33: 1.0
   Corner 37: 1.0
```

Road Strategy Variation

In [13]:

```
using JuMP, Clp, NamedArrays, Gurobi

#The possible resources
resources = [:wood, :wheat, :brick, :sheep, :ore, :nothing]

#Each time
time = [1, 2, 3]

#The values for resources over each time
resource_values = NamedArray([8 4 8 4 1 0; 5 7 5 2 3 0; 3 8 3 2 8 0], (time, resources), ("Time", "Resources"))

#Every possible corner
corners = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,

#The possible resources on each corner
resource_per_corner = NamedArray([:nothing :nothing :sheep; :nothing :nothing :sheep; :nothing :nothing :wheat; :nothing

#The chance of getting a resource on each corner
chances_per_corner = NamedArray([0 0 5; 0 0 3; 0 0 2; 0 0 5; 0 5 3; 0 3 2; 0 2 0; 0 5 3; 5 3 2; 3 2 0; 2 0 1; 0 0 3; 5 3

#The edges connecting each corner
edges = [[1,4], [1,5], [2,5], [2,6], [6,3], [3,7], [4,8], [5,9], [6, 10], [7,11], [8,12], [8,13], [9, 13], [9,14], [10,14

#Every possible harbor
harbor = [:anything, :brick, :wood, :sheep, :ore, :wheat]

#The values for each harbor
harbor_values = Dict(zip(harbor, [7, 4, 4, 4, 4, 4]))

#The corners that have harbors
harbor_corners = [[2, :anything], [5, :anything], [3, :anything], [7, :anything], [8, :sheep], [12, :sheep], [16, :brick]

#Initializing each array
list1 = zeros(54,6)
```

```
list2 = zeros(54,6)
list3 = zeros(54,6)

#The number of resources for time 1
resources_time_1 = NamedArray(list1, (corners, resources), ("Corners", "Amount of Resources"))

#The number of resources for time 1
resources_time_2 = NamedArray(list2, (corners, resources), ("Corners", "Amount of Resources"))

#The number of resources for time 1
resources_time_3 = NamedArray(list3, (corners, resources), ("Corners", "Amount of Resources"))

#Finds the amount of each resource for each time. Not including harbors
for i in 1:54
    for j in 1:3
        resources_time_1[i, resource_per_corner[i, j]] = resource_values[1, resource_per_corner[i, j]]*chances_per_corner
        resources_time_2[i, resource_per_corner[i, j]] = resource_values[2, resource_per_corner[i, j]]*chances_per_corner
        resources_time_3[i, resource_per_corner[i, j]] = resource_values[3, resource_per_corner[i, j]]*chances_per_corner
    end
end

#Finds the amount of additional resources that each harbor brings
for h in harbor_corners

    #This is the amount of resources that we "trade in" for each harbor
    total_resource_to_be_subtracted_1 = 0
    total_resource_to_be_subtracted_2 = 0
    total_resource_to_be_subtracted_3 = 0
    for j in resources
        total_resource_to_be_subtracted_1 += resources_time_1[h[1], j]
        total_resource_to_be_subtracted_2 += resources_time_2[h[1], j]
        total_resource_to_be_subtracted_3 += resources_time_3[h[1], j]
    end

    #Adding the amount of of additional resources that the harbor brings in
    #Does not include :anything
    if h[2] != :anything
        resources_time_1[h[1], h[2]] += harbor_values[h[2]]
        resources_time_2[h[1], h[2]] += harbor_values[h[2]]
        resources_time_3[h[1], h[2]] += harbor_values[h[2]]
    end

    #Adding the amount of of additional resources that the harbor brings in
    #Includes :anything
    if h[2] == :anything
        for r in resources
```

```
            resources_time_1[h[1], r] += harbor_values[h[2]]/5
            resources_time_2[h[1], r] += harbor_values[h[2]]/5
            resources_time_3[h[1], r] += harbor_values[h[2]]/5
        end
    end

    #The amount of total resources that we lose for the harbor
    #This value is stored in the row of :nothing for convinience
    resources_time_1[h[1], :nothing] = -(total_resource_to_be_subtracted_1*0.3)
    resources_time_2[h[1], :nothing] = -(total_resource_to_be_subtracted_2*0.3)
    resources_time_3[h[1], :nothing] = -(total_resource_to_be_subtracted_3*0.3)
end

#The total resources over all the times
total_resources = zeros(54)
for i in 1:54
    for j in resources
        total_resources[i] += resources_time_1[i, j]
        total_resources[i] += resources_time_2[i, j]
        total_resources[i] += resources_time_3[i, j]
    end
end

m = Model(Gurobi.Optimizer)

#Decision Variable x for each corner
@variable(m, x[1:54])
;

#Constraint on x[1:54]

#x has to be 0 or 1
@constraint(m, x_one_zero[i in 1:54], x[i]^2 == x[i])

#The total number of corners chosen is 3
@constraint(m, x_choose_three, sum(x[i] for i in 1:54) == 3)

#There has to be at least 1 harbor
@constraint(m, x_at_least_one_harbor, sum(x[i]*resources_time_1[i, :nothing]+x[i]*resources_time_3[i, :nothing]+x[i]*resc
;

#Material Constraints

#Constraint for amount of wood
@constraint(m, wood, sum(resources_time_1[i, :wood]*x[i] + resources_time_2[i, :wood]*x[i] + resources_time_3[i, :wood]*x
```

```
#Constraint for amount of brick
@constraint(m, brick, sum(resources_time_1[i, :brick]*x[i] + resources_time_2[i, :brick]*x[i] + resources_time_3[i, :bric
;

#Distance Constraints
for i in edges
    @constraint(m, x[i[1]]+ x[i[2]] <= 1)
end


#Objective Function
@objective(m, Max, sum(x[i]*total_resources[i] for i in 1:54))
;

set_optimizer_attribute(m, "NonConvex", 2)
optimize!(m)


println("Solution")
println("The Best Corners To Place Are:")
for i in 1:54
    if value(x[i])==1
        println("   Corner ", i, ": ", value(x[i]))
    end
end
```

```
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 76 rows, 54 columns and 267 nonzeros
Model fingerprint: 0x587b9c80
Model has 54 quadratic constraints
Coefficient statistics:
  Matrix range     [1e+00, 9e+01]
  QMatrix range    [1e+00, 1e+00]
  QLMatrix range   [1e+00, 1e+00]
  Objective range  [1e+01, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e-02, 1e+02]

Continuous model is non-convex -- solving as a MIP.

Presolve time: 0.00s
Presolved: 238 rows, 54 columns, 375 nonzeros
Presolved model has 54 bilinear constraint(s)
Variable types: 54 continuous, 0 integer (0 binary)
```

```
Root relaxation: objective 5.198103e+02, 46 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  519.81030    0    3          -  519.81030      -     -    0s
     0     0  519.81030    0    3          -  519.81030      -     -    0s
     0     0  519.81030    0    3          -  519.81030      -     -    0s
     0     0  519.65862    0   11          -  519.65862      -     -    0s
     0     0  519.65862    0   11          -  519.65862      -     -    0s
     0     2  519.65862    0   11          -  519.65862      -     -    0s
*   56    14              11     417.9000000  484.98126  16.1%   2.3    0s
*   61    14              10     455.9000000  484.98126  6.38%   2.2    0s

Cutting planes:
  RLT: 3

Explored 105 nodes (274 simplex iterations) in 0.06 seconds
Thread count was 4 (of 4 available processors)

Solution count 2: 455.9 417.9

Optimal solution found (tolerance 1.00e-04)
Best objective 4.559000000000e+02, best bound 4.559000000000e+02, gap 0.0000%
Solution
The Best Corners To Place Are:
   Corner 37: 1.0
   Corner 45: 1.0
   Corner 54: 1.0
```

Development Card Strategy Variation

In [14]:

```
using JuMP, Clp, NamedArrays, Gurobi

#The possible resources
resources = [:wood, :wheat, :brick, :sheep, :ore, :nothing]

#Each time
time = [1, 2, 3]

#The values for resources over each time
resource_values = NamedArray([8 4 8 4 1 0; 5 7 5 2 3 0; 3 8 3 2 8 0], (time, resources), ("Time", "Resources"))

#Every possible corner
corners = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,

#The possible resources on each corner
```

```
resource_per_corner = NamedArray([:nothing :nothing :sheep; :nothing :nothing :sheep; :nothing :nothing :wheat; :nothing

#The chance of getting a resource on each corner
chances_per_corner = NamedArray([0 0 5; 0 0 3; 0 0 2; 0 0 5; 0 5 3; 0 3 2; 0 2 0; 0 5 3; 5 3 2; 3 2 0; 2 0 1; 0 0 3; 5 3

#The edges connecting each corner
edges = [[1,4], [1,5], [2,5], [2,6], [6,3], [3,7], [4,8], [5,9], [6, 10], [7,11], [8,12], [8,13], [9, 13], [9,14], [10,14

#Every possible harbor
harbor = [:anything, :brick, :wood, :sheep, :ore, :wheat]

#The values for each harbor
harbor_values = Dict(zip(harbor, [7, 4, 4, 4, 4, 4]))

#The corners that have harbors
harbor_corners = [[2, :anything], [5, :anything], [3, :anything], [7, :anything], [8, :sheep], [12, :sheep], [16, :brick]

#Initializing each array
list1 = zeros(54,6)
list2 = zeros(54,6)
list3 = zeros(54,6)

#The number of resources for time 1
resources_time_1 = NamedArray(list1, (corners, resources), ("Corners", "Amount of Resources"))

#The number of resources for time 1
resources_time_2 = NamedArray(list2, (corners, resources), ("Corners", "Amount of Resources"))

#The number of resources for time 1
resources_time_3 = NamedArray(list3, (corners, resources), ("Corners", "Amount of Resources"))

#Finds the amount of each resource for each time. Not including harbors
for i in 1:54
    for j in 1:3
        resources_time_1[i, resource_per_corner[i, j]] = resource_values[1, resource_per_corner[i, j]]*chances_per_corner
        resources_time_2[i, resource_per_corner[i, j]] = resource_values[2, resource_per_corner[i, j]]*chances_per_corner
        resources_time_3[i, resource_per_corner[i, j]] = resource_values[3, resource_per_corner[i, j]]*chances_per_corner
    end
end

#Finds the amount of additional resources that each harbor brings
for h in harbor_corners

    #This is the amount of resources that we "trade in" for each harbor
    total_resource_to_be_subtracted_1 = 0
    total_resource_to_be_subtracted_2 = 0
```

```
    total_resource_to_be_subtracted_3 = 0
    for j in resources
        total_resource_to_be_subtracted_1 += resources_time_1[h[1], j]
        total_resource_to_be_subtracted_2 += resources_time_2[h[1], j]
        total_resource_to_be_subtracted_3 += resources_time_3[h[1], j]
    end

    #Adding the amount of of additional resources that the harbor brings in
    #Does not include :anything
    if h[2] != :anything
        resources_time_1[h[1], h[2]] += harbor_values[h[2]]
        resources_time_2[h[1], h[2]] += harbor_values[h[2]]
        resources_time_3[h[1], h[2]] += harbor_values[h[2]]
    end

    #Adding the amount of of additional resources that the harbor brings in
    #Includes :anything
    if h[2] == :anything
        for r in resources
            resources_time_1[h[1], r] += harbor_values[h[2]]/5
            resources_time_2[h[1], r] += harbor_values[h[2]]/5
            resources_time_3[h[1], r] += harbor_values[h[2]]/5
        end
    end

    #The amount of total resources that we lose for the harbor
    #This value is stored in the row of :nothing for convinience
    resources_time_1[h[1], :nothing] = -(total_resource_to_be_subtracted_1*0.3)
    resources_time_2[h[1], :nothing] = -(total_resource_to_be_subtracted_2*0.3)
    resources_time_3[h[1], :nothing] = -(total_resource_to_be_subtracted_3*0.3)
end

#The total resources over all the times
total_resources = zeros(54)
for i in 1:54
    for j in resources
        total_resources[i] += resources_time_1[i, j]
        total_resources[i] += resources_time_2[i, j]
        total_resources[i] += resources_time_3[i, j]
    end
end

m = Model(Gurobi.Optimizer)

#Decision Variable x for each corner
@variable(m, x[1:54])
```

```
;

#Constraint on x[1:54]

#x has to be 0 or 1
@constraint(m, x_one_zero[i in 1:54], x[i]^2 == x[i])

#The total number of corners chosen is 3
@constraint(m, x_choose_three, sum(x[i] for i in 1:54) == 3)

#There has to be at least 1 harbor
@constraint(m, x_at_least_one_harbor, sum(x[i]*resources_time_1[i, :nothing]+x[i]*resources_time_3[i, :nothing]+x[i]*resc
;

#Material Constraints

#Constraint for amount of sheep
@constraint(m, sheep, sum(resources_time_1[i, :sheep]*x[i] for i in 1:54) + sum(resources_time_2[i, :sheep]*x[i] for i in

#Constraint for amount of ore
@constraint(m, ore, sum(resources_time_1[i, :ore]*x[i] for i in 1:54)+sum(resources_time_2[i, :ore]*x[i] for i in 1:54)+s

#Constraint for amount of wheat
@constraint(m, wheat, sum(resources_time_1[i, :wheat]*x[i] for i in 1:54)+sum(resources_time_2[i, :wheat]*x[i] for i in 1
;

#Distance Constraints
for i in edges
    @constraint(m, x[i[1]]+ x[i[2]] <= 1)
end


#Objective Function
@objective(m, Max, sum(x[i]*total_resources[i] for i in 1:54))
;

set_optimizer_attribute(m, "NonConvex", 2)
optimize!(m)


println("Solution")
println("The Best Corners To Place Are:")
for i in 1:54
    if value(x[i])==1
        println("   Corner ", i, ": ", value(x[i]))
```

```
    end
end
```

```
Academic license - for non-commercial use only
Academic license - for non-commercial use only
Gurobi Optimizer version 9.0.3 build v9.0.3rc0 (win64)
Optimize a model with 77 rows, 54 columns and 283 nonzeros
Model fingerprint: 0xd814d5ba
Model has 54 quadratic constraints
Coefficient statistics:
  Matrix range     [1e+00, 1e+02]
  QMatrix range    [1e+00, 1e+00]
  QLMatrix range   [1e+00, 1e+00]
  Objective range  [1e+01, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e-02, 8e+01]

Continuous model is non-convex -- solving as a MIP.

Presolve time: 0.00s
Presolved: 239 rows, 54 columns, 391 nonzeros
Presolved model has 54 bilinear constraint(s)
Variable types: 54 continuous, 0 integer (0 binary)

Root relaxation: objective 4.829874e+02, 50 iterations, 0.00 seconds

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0  482.98741    0    3          -  482.98741      -     -    0s
     0     0  482.98741    0    3          -  482.98741      -     -    0s
     0     2  482.98741    0    3          -  482.98741      -     -    0s
*   29     8               9     390.9000000  476.97864  22.0%   1.4    0s
*   47     8               8     415.9000000  445.92308  7.22%   1.4    0s
*   61     1               3     430.6000000  436.24251  1.31%   1.5    0s

Explored 63 nodes (147 simplex iterations) in 0.04 seconds
Thread count was 4 (of 4 available processors)

Solution count 3: 430.6 415.9 390.9

Optimal solution found (tolerance 1.00e-04)
Best objective 4.306000000000e+02, best bound 4.306000000000e+02, gap 0.0000%
Solution
The Best Corners To Place Are:
   Corner 8: 1.0
   Corner 37: 1.0
   Corner 46: 1.0
```

4. Results and discussion

4.A. Dataset 1

The corners chosen by our model are circled in green. The model in general adheres to our strategy of trying to diversify which resources we harvest, versus monopolizing a resource in hopes of being able to trade it in for others. All 5 resources are represented in the choices the model has made, making these three corners especially valuable for procuring a variety of resources. This dataset assumed that the player it is choosing for is going first, allowing them to choose any potential corner on the map. This obviously allows for the model to choose the most optimal corners on the board.

4.B. Dataset 2

Our second dataset is a bit different. We placed some settlements already on the board to simulate when the player is not going first.

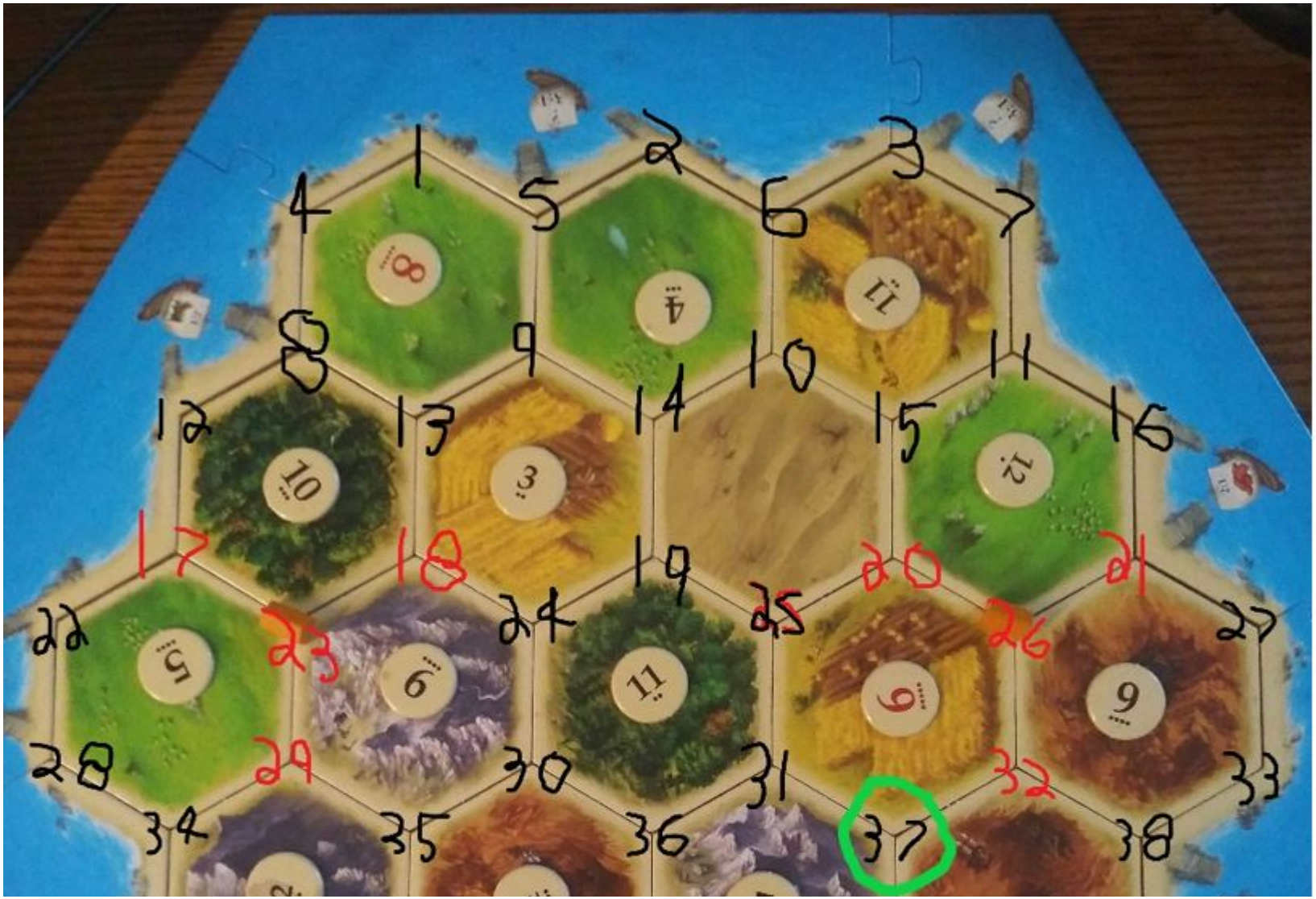
4.B.I. General Strategy

The chosen corners are again circled in green and the corners which are unavailable to our model are in red text. Certainly, choosing a corner like 33, with access to only one resource tile, seems suboptimal at first. However, brick is a highly valuable resource in the early

game, and the tile 33 is in contact with has the highest chance to be rolled out of the brick tiles on the board. All resources are again represented by our solution, allowing the player to collect a diverse set of resources throughout the game.

4.B.II. Early Game Strategy

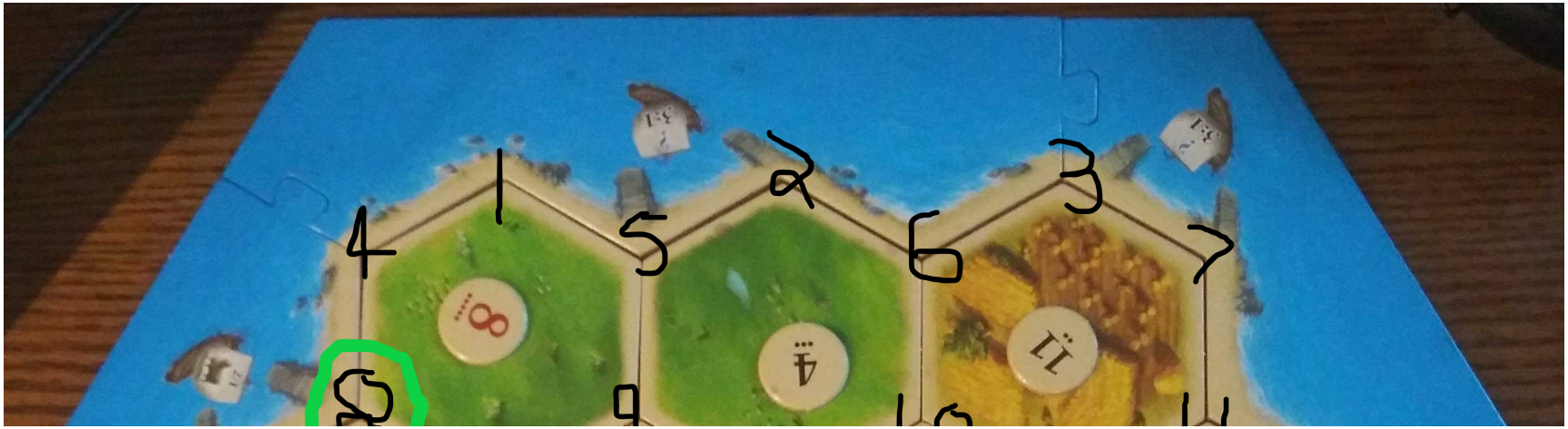
Additionally, we manipulated the model to use less general strategies. For our third solution, we choose to search for the corners that would optimize the production of resources needed to produce roads (wood and brick). Roads are very powerful in the early game of Catan, allowing a player to expand their influence quickly. Additionally, bonus points are awarded at the end of the game to the player that possesses the longest road.

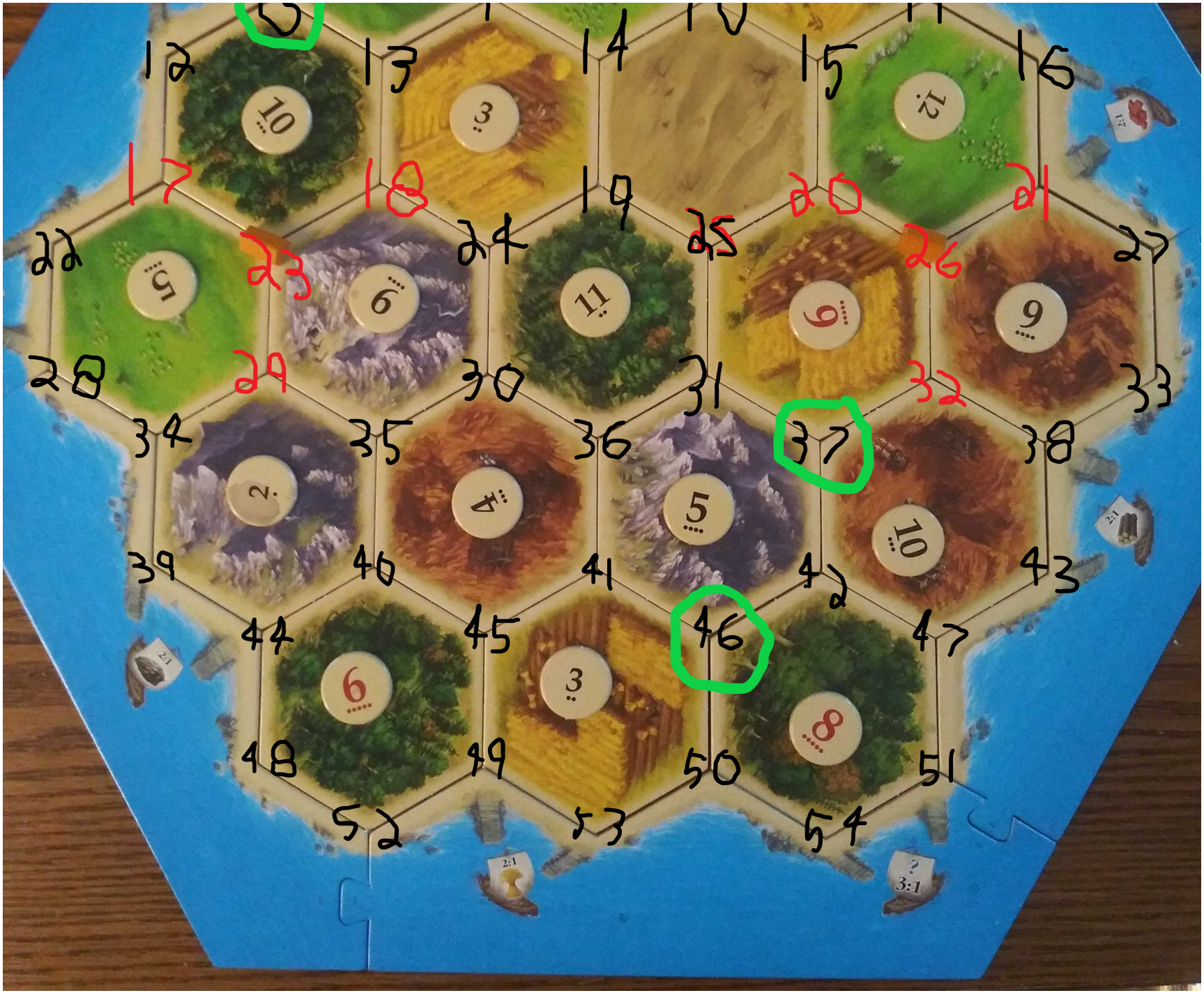


The model's choices are again highlighted in green. We ran this on dataset 2. Corners attached to wood and brick tiles are prioritized, with tiles that give the most resources being chosen first.

4.B.III. Development Card Strategy

Finally, we manipulated the model to choose tiles that produce the resources required for development cards (ore, sheep, and wheat). Development cards have a wide variety of effects that offer advantages to the player who has purchased them, such as being able to build roads for free or stealing resources from other players.




12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
10
3
12
5
6
11
9
9
2
4
5
10
6
3
8
2:1
2:1
2:1
?
3:1

This variation of the model was also run on dataset 2. It prioritizes sheep, ore, and wheat tiles.

4.C. Discussion

While our model seeks to look at settlement placement objectively, choosing those tiles that meet our resource diversity constraints and yield the highest amount of resources on average, we did make assumptions that if altered would affect how the model chooses its results. Most obviously, the weight we gave to certain resources in terms of value between early, mid, and late game are not based on any objective data. These parameters were chosen to fall in line with which structures are most useful for a general strategy in different stages of the game. However, if one were to want to pursue a far different strategy than what we outlined, some resources may appear more or less valuable than how we weighted them in our model.

5. Conclusion

Through the development of our model, we've found the optimization of settlement placement in Settlers of Catan can be done via integer programming. Given a hypothetical gameboard, we can choose the best combination of settlement locations to fit a variety of strategies. Of course, these settlement placements don't translate directly to the accumulation of resources, as which tiles are rolled is entirely random. They do however provide the best chance of harvesting those resources given a certain strategy.

The performance of our model may be improved through the use of regularization constraints. However, due to the relatively small and constant problem size, run time in reality is rather short.

Because we focused on one aspect of relatively complex game, there are many more mechanics that could either be added to our own model or modelled separately, such as resource management, choosing optimal road routes, or choosing the optimal placement of cities. Optimal city placement, for example, is a problem very similar to settlement placement. Cities are structures that must replace already existing settlements and provide double the points and double the resources from each of the tiles they are in contact with. Since the choices of where to place cities is limited to where a player's settlements are, the problem size would be smaller than our model while still using the same basic strategy of choosing the highest yield tiles. Additionally, since Catan is turn based, with the layout of the board changing almost every turn through player interaction, our model approaches settlement placement one board state at a time. This could be expanded to work more efficiently between changes in the board state such that it could be used effectively for a whole game.

The actual effectiveness of approaching settlement placement as we have in this model seems sound in theory, but due to the inefficiency of building datasets we weren't able to use the model to influence player decision in an actual game. Ideally, we would play through games using the results of our model to test the effectiveness of our approach. In practice, there are likely subjective choices we made in this model that could be tuned through this testing.