GRP 10: Project X

Andrew Gault (22xjs5)

Harsh Shrivastav (22dd37)

Nandan Bhut (22gp44)

Leo Toner Millet (22jxt)

Course Modelling Project

CISC/CMPE 204

Logic for Computing Science

December 7, 2024

Abstract

Welcome to Project X. To provide a brief context, on our first colonization mission to "Post Tenebras Spero Lucem", we lost many crew members on the long and arduous journey. But fear not, for we have a developed a model that will scale this journey once more. You, as the user and captain of this fine rocket, will use experimental technology to summon obstacles the rocket will encounter throughout the mission, and the rocket's native SAT Solver System (SATSS) will provide the optimal beacon placements to save as many crew members as possible.

The model generates, visually, the journey that is split up into three stages (insert images for this). These stages are made of grids, false values represent empty space that the rocket can move to, and true values represent planets, space objects, and people (of different colors for differentiation). The rocket will follow a fixed path, and given the layout of the grids, the model will decide where to place at most 6 beacons across all stages to save as many people as possible.

To achieve this with proposition logic, our main constraints were based on the beacons and the people. Each person has a certain defined radius that they can reach, stored in the code as reachability. The solver looks at the reachable cells, and will further decide if it should put a beacon on it depending on how many people can be saved via that one beacon.

Propositions

These are the propositions used in the encoding:

- Beacon(x, y, grid): places a beacon at a point on the outlined grid and position (y, x)
- SpaceObject (x, y, grid, P): places a space object outlined by the users input for (x, y, grid) and sets it to True (P).
- Reachable(x, y, grid): Sets a cell, (y, x), on the defined grid to True (cell is reachable). These cells are defined as a 3x3 grid around each cell in the rockets path.
- PlanetCell (x, y, grid, P): Sets a cell (y, x) on the defined grid to True (P) indicating that a planet is present.
- Person(x, y, grid, P): Sets a cell on the defined grid at y, x to True (P), indicating that a person is present on that cell.
- Person-Radius: Defines a 3x3 grid around every person proposition that represents their reach.

Constraints

The constraints in this model are mainly position based as the solver solves for beacon placements given a set of stages and then how to optimize them. These involve positions for reachable, beacons, person, and planet cells.

General Constraints

- We cannot have more than one object in the same cell. In the code, this was written as:

$$\begin{aligned} \text{PlanetCell} &\rightarrow \neg\text{Beacon} \\ \text{SpaceObject} &\rightarrow \neg\text{Beacon} \\ \text{Person} &\rightarrow \neg\text{Beacon} \end{aligned}$$

Beacon

- A beacon can't save no people, i.e. if there is no person within its reach, it cannot be placed there.

$$\neg(\text{Person-Radius}) \rightarrow \neg\text{Beacon}$$

- If we add a person, planet or a space object to a cell, we cannot place a beacon there.

$$(\text{SpaceObject} \vee \text{PlanetCell} \vee \text{Person}) \rightarrow \neg(\text{Beacon})$$

- A beacon must save at least 1 more person, i.e. if a person is saved by a beacon, they cannot be the only person saved by a different beacon. In terms of grid based logic, we want to add beacons where the reach of people radii overlap (for optimization). These overlaps could involve multiple people as well. If a cell has multiple overlaps, the solver will prefer to add the beacon there. So after getting the position of these overlaps, we do a similiar thing to before:

$$((\text{Person1-Radius}) \wedge (\text{Person2-Radius}) \text{ ... }) \wedge \text{Reachable} \rightarrow \text{Beacon}$$

- A beacon must be placed on a reachable position, i.e. it cannot be placed outside of the grid or on positions that have no people radii on it.

$$\text{Beacon} \rightarrow (\text{Reachable} \wedge \text{Person-Radius})$$

Model Exploration

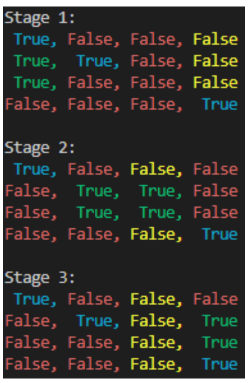
Beginning of the model

We began our model by creating three independent stages for the rocket to complete its journey. Stage one: takeoff, Stage two: use the planet as an orbital assist, and Stage three: landing. First, we started with a generic grid with specific spaces set aside for planets which was our initial proposition. The rocket would then go through the grid which consists of cells to find the most efficient route to its final destination (another planet). We had fuel as one of our primary constraints, which we quantified using binary systems which were made up of propositions. Each of these actions-moving from cell to cell or orbital assists-required a certain quantity of fuel. This was suppose to show us if the amount of fuel that was entered by the user was enough to complete the journey.

Grid System

The most fundamental part of this model was developing the grid system that the rocket would move on. Thinking about it in terms of propositional logic, we had to use True's and Falses to base the grid system around, so when it came to dealing with objects and movements on the system, it would be constraint based. The grid is developed based on the radius of the planets, with the planets centered in different areas depending on the stage (representing take off (left-centered), orbital assist (middle) and landing (right-centered)). The rocket moves around these planets, through checkpoints, people, and placed space objects. To make all these moving parts fairly clear and distinguishable, our very important (and sometimes very annoying) debug-print function was quite useful.

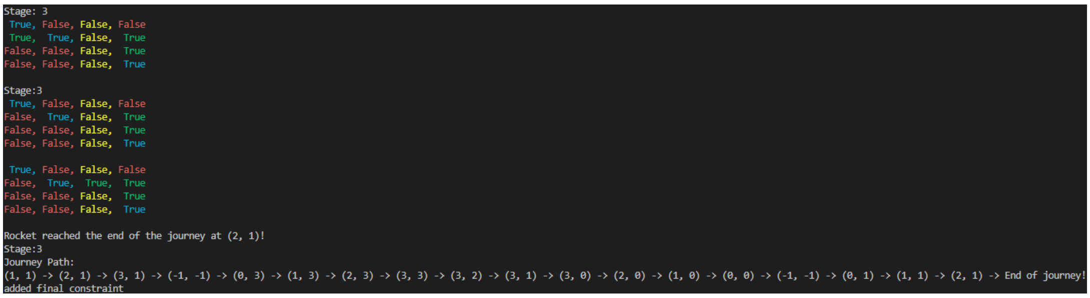
Planets represented in green as well as the rocket, checkpoints were yellow, and people are blue. We also had to make sure to do this for our solution that the SAT solver provided by making the beacons stand out, as that is the primary solution the model is providing, even though the solver needs to check every reachable position (which is the reason for the large print after the solver solves the model).



Each cell in the grid is set to false on default (unless a person or space object is generated there) and so the rocket checks if a cell is available based on it being false to move to. Leading to our rocket movement system. If we were to start this project again, we would make the movement and grid system more constraint based. So rather than python code doing the movement, the solver would do it and do like a pathfinder approach based on gravity, planets, and speed. This was a major forkway upon deciding for this or the beacon system. Given the time we had, the beacon system made more sense.

Checkpoints and Movement

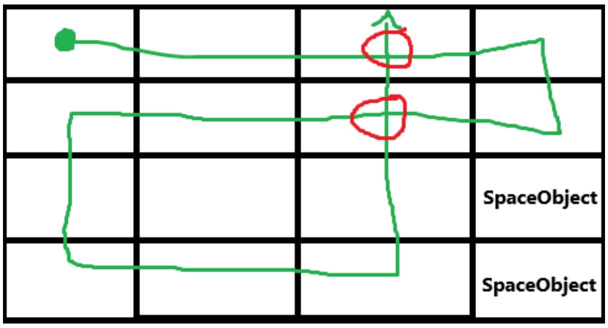
Working on the rocket's movement was a major challenge at first; we considered using propositional checkpoints as a way for the rocket to follow an efficient path, these checkpoints would be added around the planet's radius. We employed a (y,x) coordinate system for rocket movement, which was more efficient than the basic (x,y) coordinates we used initially. This was because the rocket was mostly going in the x direction and would have a greater spatial locality than looping over the next outer array each time. We had x, y as a 2D array, so having x come last and be in the inner most array made the most sense and was more efficient. Furthermore, the rockets movement dynamic also changed. Initially, the rocket would always check the cell to the right, and then move down if there was an obstacle in the way. Since this was a general python approach, to make it more compatible with our constraints, we changed it so it uses a direction system. The rocket and its checking system would all turn together, which allowed us to check if it finds itself in a loop, which is a required condition for an unsolvable model, i.e. imagine if there was a wall of space objects, the rocket would loop around and check if it has visited two positions it already has had before, and thus it recognize its in a loop.



Bug For Movement

We initially tried to identify if the rocket was caught in an infinite loop by checking if the current point and the previous point had already appeared earlier in the journey. However, we realized that this did not make sense conceptually because it doesn't tell us that the rocket was stuck in a loop by just checking the two positions. The picture below shows how it could easily be that it can pass through the same points but at completely different directions, therefore the rocket could be going somewhere else but still revisits some point. hence this approach is not enough to conclude that it's stuck in some kind of loop. To address this, we later implemented a stack (mistakenly referred to as a queue) to track whether the current vector had been encountered before, which successfully resolved the issue. since this time we are taking into account the direction we can see exactly where the rocket might be going and safely say if its stuck in a loop or changing course.

Furthermore, for stage 2, the direction system required a major rework. Essentially, the direction system worked with moving the rocket 90 degrees to move it up, but if it encountered space object in that last column, it would be stuck in a loop there instead of backtracking all the way around to (0,0). This is because to do that, it would have to move -90 degrees in that one specific instance, which is too complicated a code in general python to look at for this model.



Improved Model

At this stage, we understood that our initial constraints were too simple, with fuel being the only important factor. In order to make the problem more interesting and challenging, we decided to add more constraints and propositions to it. After some brainstorming, we implemented the idea of a rocket's journey into a more interactive game.

The new mission, in this revised scenario, involves not just the journey of the rocket but saving people scattered on a grid. People are stranded in space, and through their movement in a grid, the rocket has to drop beacons to save them. This new mission brings in strategic challenges of how best to use the fuel, drop the beacons, and find the optimum path to rescue the maximum number of people.

This helped us complicate our model more with whole new set of constraints and propositions. Some of the added newer constraints were:

Beacons could not be deployed in areas where no humans were within their radius.

Beacons had to be placed efficiently to save the maximum number of humans. Like outlined in our constraints, this is done by looking at reachable cells that have overlap of person radii. Essentially, the code looks at where the people are and their respective reach, and then searches for conjunctions (overlaps) and tells the solver to put beacons there for optimization. The code for this was as such:

```
for a in range(len(people_positions)):
    self_pos = people_positions[a]
    conjunct_pos = []
    for b in range(len(people_positions)):
        other_pos = people_positions[b]

        # Early check to lower average computation
        if (self_pos != other_pos and self_pos[2] == other_pos[2] # If not the same person, if in the same grid
        and other_pos[0] >= self_pos[0] - BEACON_RANGE * 2 and other_pos[0] <= self_pos[0] + BEACON_RANGE * 2 # If y range overlaps
        and other_pos[1] >= self_pos[1] - BEACON_RANGE * 2 and other_pos[1] <= self_pos[1] + BEACON_RANGE * 2 # If x range overlaps
        ):
            # Find location(s) of conjunction(s)
            for x_self in range(BEACON_RANGE * 2 + 1):
                for y_self in range(BEACON_RANGE * 2 + 1):
                    for x in range(BEACON_RANGE * 2 + 1):
                        for y in range(BEACON_RANGE * 2 + 1):
                            if (other_pos[1] - BEACON_RANGE + x == self_pos[1] - BEACON_RANGE + x_self # What x value overlaps?
                            and other_pos[0] - BEACON_RANGE + y == self_pos[0] - BEACON_RANGE + y_self # What y value overlaps?
                            ):
                                if((self_pos[0] - BEACON_RANGE + y_self, self_pos[1] - BEACON_RANGE + x_self, self_pos[2]) in reachable):
                                    conjunct_pos.append((self_pos[0] - BEACON_RANGE + y_self, self_pos[1] - BEACON_RANGE + x_self, self_pos[2

    # If there is a conjunction, set all non-conjunction cells relative to self_pos to ~Beacon(...)
    if len(conjunct_pos) != 0:
        for x in range(BEACON_RANGE * 2 + 1):
            for y in range(BEACON_RANGE * 2 + 1):
                if ((self_pos[0] - BEACON_RANGE + y, self_pos[1] - BEACON_RANGE + x, self_pos[2]) not in conjunct_pos):
                    E.add_constraint(~Beacon(x, y, self_pos[2]))
```

Human Placement

The human placement on the grid was to simulate a rescue scenario and to have a predictable placement of humans we created an algorithm that placed humans on the grid randomly but while also maintaining a pattern. This made it so that the placement was not entirely random but followed a predictable logic that allowed the beacon placement to be more efficient since the human placement follows a pattern that eliminates things like there being all the humans placed in one certain cluster or overlaps. This would be useful for the user to notice patterns of where the humans might be and see what spots could be the most optimal for beacon placement.

Removing fuel

At this point, we removed the fuel constraint from our model. Fuel was put into the model initially as a main factor to introduce some level of realism and complexity in the model. However, we found that the SAT solver wasn't using this constraint effectively, nor did it provide any meaningful contribution to the process of solving the problem. As we also shifted our model to a game-like framework, unrestricted movement became a necessity to allow for gameplay dynamics and flexibility. The constraint became incompatible with the new model, as it made the rocket less capable of exploring and traversing the grid. Given that the new model favored gameplay over the strict following of realistic constraints, the fuel system became redundant and pretty much useless to this model.

Jape Proofs

Proof 1:

Synopsis: Given that all cells (positions) are Planets, there are no ValidLocations for any Beacon.

Translation: Given 'no two objects can be in the same position'; and every object in the grid is a PlanetCell; and if every object is one type of object, the other type of object must always be false (one or the other); and if the 'other object' (S1) is False, then every location must be a PlanetCell; and if one proposition of the left side of the implication in 'A Beacon must contain at least one person within its saving radius' (shown in First-Order Extension) is False, then it is not a ValidLocation. Which was proven to be semantically equivalent to ¬ Q1(x3, x4), meaning there are no ValidLocations for any Beacon. [Every x1, x2, ect. has been matched up to the same equivalent actual i (i.e. two different x1's are linked to actual i, while an x5 is linked to actual i2) to make it more valid.]

Legend: S(x1) = Object(a) (We will let this object be of type PlanetCell); S1(x2) = Object(b) (Person(c)); Q(x3) = Location(c); P(x1, x3) = Pos(a, c); P(x2, x3) = Pos(b, c); S2(x4) = Beacon(a); R(x2, x5) = Range(c, b); P(x4, x3) = Pos(a, d); R1(x4, x2) = WithinRange(d, c); Q1(x3, x4) = ValidLocation(d, a).

1:	∀x1.∀x2.∀x3.((S(x1)∧S1(x2)∧Q(x3)∧P(x1,x3))→¬P(x2,x3)), ∀x1.(S(x1)), ∀x1.∀x2.(¬S(x1)∨¬S1(x2)), ∀x1.∀x2.∀x3.(¬S1(x2)→(Q(x3)∧P(x1,x3)))	premises
2:	∀x4.∀x5.∀x2.∀x3.(¬(S2(x4)∧R(x2,x5)∧S1(x2)∧Q(x3)∧(P(x4,x3)∧R1(x3,x2)))→¬Q1(x3,x4)), actual i, actual i1, actual i2	premises
3:	S(i)	∀ elim 1.2,2.2
4:	∀x2.∀x3.((S(i)∧S1(x2)∧Q(x3)∧P(i,x3))→¬P(x2,x3))	∀ elim 1.1,2.2
5:	∀x3.((S(i)∧S1(i1)∧Q(x3)∧P(i,x3))→¬P(i1,x3))	∀ elim 4,2.3
6:	∀x2.(¬S(i)∨¬S1(x2))	∀ elim 1.3,2.2
7:	¬S(i)∨¬S1(i1)	∀ elim 6,2.3
8:	∀x2.∀x3.(¬S1(x2)→(Q(x3)∧P(i,x3)))	∀ elim 1.4,2.2
9:	∀x3.(¬S1(i1)→(Q(x3)∧P(i,x3)))	∀ elim 8,2.3
10:	actual i3	assumption
11:	(S(i)∧S1(i1)∧Q(i3)∧P(i,i3))→¬P(i1,i3)	∀ elim 5,10
12:	¬S1(i1)→(Q(i3)∧P(i,i3))	∀ elim 9,10
13:	actual i4	assumption
14:	∀x5.∀x2.∀x3.(¬(S2(i4)∧R(x2,x5)∧S1(x2)∧Q(x3)∧(P(i4,x3)∧R1(x3,x2)))→¬Q1(x3,i4))	∀ elim 2.1,13
15:	∀x2.∀x3.(¬(S2(i4)∧R(x2,i2)∧S1(x2)∧Q(x3)∧(P(i4,x3)∧R1(x3,x2)))→¬Q1(x3,i4))	∀ elim 14,2.4
16:	∀x3.(¬(S2(i4)∧R(i1,i2)∧S1(i1)∧Q(x3)∧(P(i4,x3)∧R1(x3,i1)))→¬Q1(x3,i4))	∀ elim 15,2.3
17:	¬(S2(i4)∧R(i1,i2)∧S1(i1)∧Q(i3)∧(P(i4,i3)∧R1(i3,i1)))→¬Q1(i3,i4)	∀ elim 16,10
18:	¬S(i)	assumption
19:	⊥	¬ elim 3,18
20:	¬Q1(i3,i4)	contra (constructive) 19
21:	¬S1(i1)	assumption
22:	Q1(i3,i4)	assumption
23:	S2(i4)∧R(i1,i2)∧S1(i1)∧Q(i3)∧(P(i4,i3)∧R1(i3,i1))	assumption
24:	S2(i4)∧R(i1,i2)∧S1(i1)∧Q(i3)	∧ elim 23
25:	S2(i4)∧R(i1,i2)∧S1(i1)	∧ elim 24
26:	S1(i1)	∧ elim 25
27:	⊥	¬ elim 26,21
28:	¬(S2(i4)∧R(i1,i2)∧S1(i1)∧Q(i3)∧(P(i4,i3)∧R1(i3,i1)))	¬ intro 23-27
29:	¬Q1(i3,i4)	→ elim 17,28
30:	⊥	¬ elim 22,29
31:	¬Q1(i3,i4)	¬ intro 22-30
32:	¬Q1(i3,i4)	∨ elim 7,18-20,21-31
33:	∀x4.(¬Q1(i3,x4))	∀ intro 13-32
34:	∀x3.∀x4.(¬Q1(x3,x4))	∀ intro 10-33

Proof 2:

Synopsis: If there does not exist a location that's within the range of a person, there are no valid locations for any Beacons to exist.

Legend: S(x1) = Beacon(a); R(x3, x2) = Range(c, b); P(x3) = Person(c); Q(y) = Location(d); P2(x1, y) = Pos(a, d); R2(y, x3) = WithinRange(d, c); Q2(y, x1) = ValidLocation(d, a).

1:	∀x1.∀x2.∀x3.(¬∃y.(S(x1)∧R(x3,x2)∧P(x3)∧Q(y)∧(P2(x1,y)∧R2(y,x3))→Q2(y,x1)))	premise
2:	actual i, actual i1	premises
3:	actual i2	assumption
4:	∀x2.∀x3.(¬∃y.(S(i2)∧R(x3,x2)∧P(x3)∧Q(y)∧(P2(i2,y)∧R2(y,x3))→Q2(y,i2)))	∀ elim 1,3
5:	∀x3.(¬∃y.(S(i2)∧R(x3,i)∧P(x3)∧Q(y)∧(P2(i2,y)∧R2(y,x3))→Q2(y,i2)))	∀ elim 4,2.1
6:	¬∃y.(S(i2)∧R(i1,i)∧P(i1)∧Q(y)∧(P2(i2,y)∧R2(y,i1))→Q2(y,i2))	∀ elim 5,2.2
7:	actual i3	assumption
8:	Q2(i3,i2)	assumption
9:	S(i2)∧R(i1,i)∧P(i1)∧Q(i3)∧(P2(i2,i3)∧R2(i3,i1))	assumption
0:	Q2(i3,i2)	hyp 8
1:	S(i2)∧R(i1,i)∧P(i1)∧Q(i3)∧(P2(i2,i3)∧R2(i3,i1))→Q2(i3,i2)	→ intro 9-10
2:	∃y.(S(i2)∧R(i1,i)∧P(i1)∧Q(y)∧(P2(i2,y)∧R2(y,i1))→Q2(y,i2))	∃ intro 11,7
3:	⊥	¬ elim 12,6
4:	¬Q2(i3,i2)	¬ intro 8-13
5:	∀y.(¬Q2(y,i2))	∀ intro 7-14
6:	∀x1.∀y.(¬Q2(y,x1))	∀ intro 3-15

Proof 3:

Synopsis: Given no two objects can be in the same position; every object is a PlanetCell; if there is only one kind of object there is none of any other; if all other objects are False, planets must be in every location: A Beacon must still contain at least one person within its saving radius remains True.

Legend: S(x1) = Object(a) (of type PlanetCell; S1(x2) = Object(b) (Person(c)); Q(x3) = Location(c); P(x1, x3) = Pos(a, c); P(x2, x3) = Pos(b, c); S2(x4) = Beacon(a); R(x2, x5) = Range(c, b); P(x4, x3) = Pos(a, d); R1(x3, x2) = WithinRange(d, c); Q1(x3, x4) = ValidLocation(d, a).

1:	∀x1.∀x2.∀x3.((S(x1)∧S1(x2)∧Q(x3)∧P(x1,x3))→¬P(x2,x3)), ∀x1.(S(x1)), ∀x1.∀x2.(¬S(x1)∨¬S1(x2)), ∀x1.∀x2.∀x3.(¬S1(x2)→(Q(x3)∧P(x1,x3))), actual i	premises
2:	∀x2.∀x3.((S(i)∧S1(x2)∧Q(x3)∧P(i,x3))→¬P(x2,x3))	∀ elim 1.1,1.5
3:	∀x2.(¬S(i)∨¬S1(x2))	∀ elim 1.3,1.5
4:	∀x2.∀x3.(¬S1(x2)→(Q(x3)∧P(i,x3)))	∀ elim 1.4,1.5
5:	S(i)	∀ elim 1.2,1.5
6:	actual i1	assumption
7:	actual i2	assumption
8:	actual i3	assumption
9:	∀x3.(¬S1(i3)→(Q(x3)∧P(i,x3)))	∀ elim 4,8
10:	¬S(i)∨¬S1(i3)	∀ elim 3,8
11:	∀x3.((S(i)∧S1(i3)∧Q(x3)∧P(i,x3))→¬P(i3,x3))	∀ elim 2,8
12:	actual i4	assumption
13:	(S(i)∧S1(i3)∧Q(i4)∧P(i,i4))→¬P(i3,i4)	∀ elim 11,12
14:	¬¬P(i3,i4)→¬(S(i)∧S1(i3)∧Q(i4)∧P(i,i4))	Theorem P→Q ⊢ ¬Q→¬P 13
15:	¬S1(i3)→(Q(i4)∧P(i,i4))	∀ elim 9,12
16:	S2(i1)∧R(i3,i2)∧S1(i3)∧Q(i4)∧(P(i1,i4)∧R1(i4,i3))	assumption
17:	S2(i1)∧R(i3,i2)∧S1(i3)∧Q(i4)	∧ elim 16
18:	S2(i1)∧R(i3,i2)∧S1(i3)	∧ elim 17
19:	S1(i3)	∧ elim 18
20:	¬S(i)	assumption
21:	⊥	¬ elim 5,20
22:	Q1(i4,i1)	contra (constructive) 21
23:	¬S1(i3)	assumption
24:	Q(i4)∧P(i,i4)	→ elim 15,23
25:	P(i,i4)	∧ elim 24
26:	Q(i4)	∧ elim 24
27:	¬Q1(i4,i1)	assumption
28:	⊥	¬ elim 19,23
29:	Q1(i4,i1)	contra (classical) 27-28
30:	Q1(i4,i1)	∨ elim 10,20-22,23-29
31:	S2(i1)∧R(i3,i2)∧S1(i3)∧Q(i4)∧(P(i1,i4)∧R1(i4,i3))→Q1(i4,i1)	→ intro 16-30
32:	∀x3.(S2(i1)∧R(i3,i2)∧S1(i3)∧Q(x3)∧(P(i1,x3)∧R1(x3,i3))→Q1(x3,i1))	∀ intro 12-31
33:	∀x2.∀x3.(S2(i1)∧R(x2,i2)∧S1(x2)∧Q(x3)∧(P(i1,x3)∧R1(x3,x2))→Q1(x3,i1))	∀ intro 8-32
34:	∀x5.∀x2.∀x3.(S2(i1)∧R(x2,x5)∧S1(x2)∧Q(x3)∧(P(i1,x3)∧R1(x3,x2))→Q1(x3,i1))	∀ intro 7-33
35:	∀x4.∀x5.∀x2.∀x3.(S2(x4)∧R(x2,x5)∧S1(x2)∧Q(x3)∧(P(x4,x3)∧R1(x3,x2))→Q1(x3,x4))	∀ intro 6-34

First-Order Extension

Describe how you might extend your model to a predicate logic setting, including how both the propositions and constraints would be updated. There is no need to implement this extension!

Objects:

- PlanetCell(x): x is a PlanetCell.

- SpaceObject(x): x is a SpaceObject.

- Beacon(x): x is a Beacon.

- Person(x): x is a Person.

- Object(x): x is one of the above objects (SpaceObject, PlanetCell, Beacon, or Person).

- Location(x): x is a position (x, y, grid).

Predicates to talk about the object:

- Reachable(x): x is reachable from the rocket.

- Range(x, y): x has range y.

- WithinRange(x, y): x is within y's range.

- Pos(z, y): z is at a location y (x, y, grid).

- ValidLocation(x, y): y is a valid location to place x.

- (x = y): x is equal to y.

- (x ≠ y): x is not equal to y.

No two objects can be in the same position at the same time.
Description: Given every object (SpaceObject, PlanetCell, Beacon, Person) that are not the same object (i.e. A SpaceObject and PlanetCell is valid, a SpaceObject and a different SpaceObject is valid, but not a SpaceObject and the exact same SpaceObject) both objects cannot be in the same position.

$$\forall a.\forall b.\exists c.((Object(a) \land Object(b) \land Location(c) \land Pos(a, c) \land a \neq b) \implies \neg(Pos(b, c)))$$

A Beacon must contain at least one person within its saving radius.

Description: For every possible Beacon saving radius, every Beacon must have at least one Person within its saving range.

$$\forall a.\forall b.\forall c.\exists d.(Beacon(a) \land Range(c, b) \land Person(c) \land Location(d)$$

$$\land(Pos(a, d) \land WithinRange(d, c)) \implies ValidLocation(d, a))$$

A Beacon must be Reachable.
Description: For a position to be able to have a Beacon placed on it, it needs to be Reachable by the rocket, which has a 3x3 grid of Reachability around it for every cell along its path. This means every location that is not Reachable must not be a ValidLocation for all Beacons.

$$\forall a.\forall b.((Beacon(a) \land Location(b) \land \neg Reachable(b)) \implies \neg ValidLocation(b, a))$$

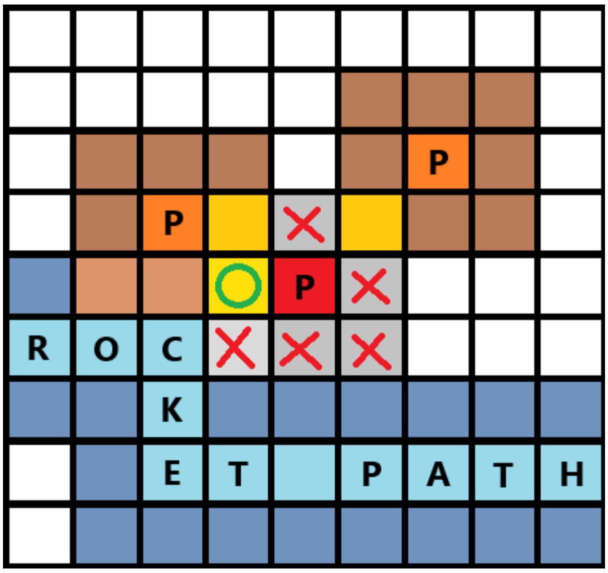
A Beacon must save at least one person that hasn't already been saved.

Description: If multiple people's ranges overlap, and at least one of the positions in the overlap is reachable, every position that is in the range of the focused person and is not in an overlap must be an invalid location for a Beacon.

This only works recursively:

$$\forall a.\forall b.\exists c.(Beacon(a) \land Person(b) \land Location(c) \land Pos(b, c) \land WithinRange(c, a) \implies \neg Reachable(b))$$

The idea behind the next two possible First Order Extension formulas to solve this problem can be easier understood through this image (Note, if there are no near people, all cells that are Reachable within focused person's range are valid. i.e. This constraint would not remove validity):



Where the darker blue around the rocket path is the Reachable range of the rocket (3x3 around each rocket path cell); the red cell is the focused Person; the two orange cells are People whose range overlaps with the focused Person (We will consider the range of a Person to be interchangeable with the range of a Beacon for the sake of these constraints, given that they are the same radius around the object, since the distance between a Person and Beacon is the same distance either direction); the brown cells are the range of the orange People; the grey cells are the range of the focused beacon; the yellow cells are the overlap (conjunction) between the radius of the focused Person and the near People; the green circle is a point where an overlap is Reachable; so since there is a Reachable overlap, all non-overlaps around the focused Person must not have a Beacon (Represented by x's through the grey cells).

This only works for two people:

$$\forall a.\forall b.\forall c.\forall d.\forall e.\forall f.((Location(a) \land Location(b) \land Person(c) \land Person(d) \land Location(e) \land Beacon(f)$$

$$\land Range(c, g) \land Range(d, g)$$

$$\land WithinRange(a, c) \land WithinRange(b, d) \land (c \neq d) \land WithinRange(e, c) \land \neg WithinRange(e, d)$$

$$\land (a = b) \land Reachable(a)) \implies (\neg ValidLocation(e, f)))$$

To have this formula work for any number of people, we need a new proposition:

- People(x, y): x is all people close to the person y.

With this, we can replace Person(d) with People(d, c), which while based on the way we defined People means that there is only one possible x in the domain per y, seems to be the simplest way to represent this problem. WithinRange still works as normal, however "y's range" will be the range of positions of all people in d.

Given that, this is the final formula:

$$\forall a.\forall b.\forall c.\forall d.\forall e.\forall f.\forall g.((Location(a) \land Location(b) \land Person(c) \land People(d, c) \land Location(e) \land Beacon(f)$$

$$\land Range(c, g) \land Range(d, g)$$

$$\land WithinRange(a, c) \land WithinRange(b, d) \land (c \neq d) \land WithinRange(e, c) \land \neg WithinRange(e, d)$$

$$\land (a = b) \land Reachable(a)) \implies (\neg ValidLocation(e, f)))$$

Here is some primarily Python code that best describes this given any number of people:

```python
# Beacon must save at least 1 more person (if one person is saved by a beacon, they cannot be the only person saved by a different beacon)
# Summary: Go out by radius * 2, if there are no people within that range, change nothing. If there are, the overlap of radii can have a beacon, so the places on the
# focused person that do not have overlap cannot have a beacon.

for a in range(len(people_positions)):
    self_pos = people_positions[a]
    conjunct_pos = []
    for b in range(len(people_positions)):
        other_pos = people_positions[b]

        # Early check to lower average computation
        if (self_pos != other_pos and self_pos[2] == other_pos[2] # If not the same person, if in the same grid
        and other_pos[0] >= self_pos[0] - BEACON_RANGE * 2 and other_pos[0] <= self_pos[0] + BEACON_RANGE * 2 # If y range overlaps
        and other_pos[1] >= self_pos[1] - BEACON_RANGE * 2 and other_pos[1] <= self_pos[1] + BEACON_RANGE * 2 # If x range overlaps
        ):
            # Find location(s) of conjunction(s)
            for x_self in range(BEACON_RANGE * 2 + 1):
                for y_self in range(BEACON_RANGE * 2 + 1):
                    for x in range(BEACON_RANGE * 2 + 1):
                        for y in range(BEACON_RANGE * 2 + 1):
                            if (other_pos[1] - BEACON_RANGE + x == self_pos[1] - BEACON_RANGE + x_self # What x value overlaps?
                            and other_pos[0] - BEACON_RANGE + y == self_pos[0] - BEACON_RANGE + y_self # What y value overlaps?
                            ):
                                if((self_pos[0] - BEACON_RANGE + y_self, self_pos[1] - BEACON_RANGE + x_self, self_pos[2]) in reachable):
                                    conjunct_pos.append((self_pos[0] - BEACON_RANGE + y_self, self_pos[1] - BEACON_RANGE + x_self, self_pos[2]))

    # If there is a conjunction, set all non-conjunction cells relative to self_pos to ~Beacon(...)
    if len(conjunct_pos) != 0:
        for x in range(BEACON_RANGE * 2 + 1):
            for y in range(BEACON_RANGE * 2 + 1):
                if ((self_pos[0] - BEACON_RANGE + y, self_pos[1] - BEACON_RANGE + x, self_pos[2]) not in conjunct_pos):
                    E.add_constraint(~Beacon(x, y, self_pos[2]))
```