



Grau en Enginyeria Informàtica

PROJECTE FINAL DE GRAU

Factorio Planner

Autor:

Pau Jimeno Román

Tutors:

Dr. Mateu Villaret Auselle

MEMÒRIA

Convocatòria:

Setembre 2024

Departament :

Arquitectura i Tecnologia de computadors

Projecte: Projecte Final de Grau
Document: Memòria
Títol: Factorio Planner
Autor: Pau Jimeno Román
Data: Setembre 2024

Estudi:
Grau en Enginyeria Informàtica
Universitat de Girona

Supervisor 1:
Dr. Mateu Villaret Auselle
Universitat de Girona
Email: mvillaret@udg.edu
Web: [Perfil UdG](#)

Índex

1	Introducció, motivacions, propòsit i objectius del projecte	1
1.1	Introducció	1
1.2	Motivacions i propòsit	2
1.3	Objectius del projecte	2
2	Estudi de viabilitat	4
2.1	Viabilitat del projecte	4
2.2	Recursos usats	5
3	Metodologia	6
4	Planificació	7
4.1	Planificació inicial del projecte	7
4.2	Seguiment de la planificació	7
4.2.1	Ús de l'API de Z3	7
4.2.2	Anàlisi del treball fet a St. Andrews	8
4.2.3	Implementació del model bàsic	8
4.2.4	Creació d'instàncies	8
4.2.5	Aplicar millors sobre el model base	9
4.2.6	Visualització de les instàncies resoltes	9
5	Marc de treball i conceptes previs	11
5.1	Problemes de satisfacció de restriccions	11
5.2	Complexitat computacional	12
5.2.1	Màquina de Turing	12
5.2.2	Classes de complexitat	12
	Classe P	12
	Classe NP i NP-complet	12
	NP-hard	13
5.3	SAT	13
5.4	SMT	14
5.5	Elements bàsics del joc Factorio	15
5.5.1	Cinta transportadora	16
5.5.2	Inseridor	16
5.5.3	Assemblador	17
	Receptes	18
5.6	Conceptes relacionats amb els elements bàsics del joc	19
5.6.1	Ruta	19
5.6.2	Tipus d'objectes	19
5.6.3	Quantitat d'objectes	19
5.7	El problema del blueprint	20

5.7.1	Exemple del problema	20
6	Requisits del sistema	23
6.1	Z3	24
6.1.1	Tipus i declaració de variables	24
6.1.2	Operadors lògics	25
6.2	Flask	25
6.2.1	Crear Endpoints usant Flask	26
6.2.2	Estructura de fitxers	26
6.2.3	Peticions des del client web	27
7	Estudis i decisions	28
7.1	Llibreria de solving	28
7.2	Llenguatge de programació	28
7.3	Interfície gràfica	28
7.4	Connexió client servidor	29
7.5	Estructura del model	29
7.6	Instàncies	30
8	Disseny del model	31
8.1	Implementació del model base	31
8.1.1	Rutes	31
Augment de ruta	31	
Decrement de ruta	32	
Entrada i sortida de les cintes	33	
Entrada i sortida dels <i>inserters</i>	34	
8.1.2	Restriccions dels <i>assemblers</i>	35
Col·lisió dels <i>assemblers</i>	36	
Link assembler collision	37	
8.1.3	Tipus d'objectes	38
Part of route	38	
Entrada i sortida d'objectes	39	
Propagació del tipus d'objectes	39	
8.1.4	Quantitat d'objectes	40
Part of route	40	
Quantitat d'entrada	41	
Propagació de la quantitat d'objectes (Cintes)	41	
Propagació de la quantitat d'objectes (Inserters)	42	
8.1.5	Receptes	43
Associar recepta	44	
Ingredients d'entrada i sortida	44	
Ràtios d'entrada	46	
Ràtio mínima	46	
Quantitat d'objectes de sortida	47	
8.2	Canvis fets al model per aproximar-lo més al comportament real del joc	47
8.2.1	Propagació de la quantitat d'objectes en inserters	48
8.2.2	Ràtios d'entrada i ràtio mínima	48
8.2.3	Quantitat d'objectes de sortida	49
8.3	Millors al model	50
8.3.1	Upper bound	50
8.3.2	Lower bound	50

8.3.3	Computar les receptes associades als assemblers	51
	Objectes i receptes	51
	Sistema d'equacions	51
	Limitar l'espai de solucions	52
	Optimització	53
8.3.4	Eliminació de simetries	53
9	Disseny del front end	56
A	Frequently Asked Questions	57
A.1	How do I change the colors of links?	57
	Bibliografia	58

Capítol 1

Introducció, motivacions, propòsit i objectius del projecte

1.1 Introducció

Factorio [1] és un joc 2D en tercera persona que se centra en l'obtenció de recursos en un planeta desolat ple d'alienígenes hostils. L'objectiu principal tracta d'obtenir suficients recursos per a enviar un coet al teu planeta d'origen. La majoria de recursos s'han de processar en fàbriques les quals amb la infraestructura adequada poden produir objectes de manera automàtica.

El joc encara està en desenvolupament i rep actualitzacions de manera freqüent. Darrere del joc s'hi troba Wube Software un estudi indie instal·lat a la República Txeca fundat l'any 2013 format per tres membres.

L'interès pel joc sorgeix arran de les mecàniques que incorpora per la creació de fàbriques i com l'objectiu principal del joc et força a optimitzar-les, ampliar-les i unir-les per generar més recursos i més complexos, d'aquí sorgeix la frase "*the factory must grow*", usada en clau d'humor per la comunitat del joc. Aquesta part del disseny de fàbriques és la que incorpora problemes d'optimització coneguts com Bin Packing, Flow i Routing. És aquí on una eina que pogués ajudar a fer dissenys òptims, parametritzables i de manera interactiva seria molt útil per la comunitat del Factorio.



FIGURA 1.1: Captura de pantalla d'una fàbrica dins el joc

1.2 Motivacions i propòsit

Des de petit que els jocs d'enginy, trencaclosques, Legos, Meccanos, cubs de Rubik... m'han apassionat i des que a la carrera vam veure la complexitat computacional que hi ha al darrere i com es poden abordar amb diferents tècniques aquesta passió s'ha desviat a com solucionar aquests problemes, optimitzar i respondre el màxim de preguntes que poden sorgir.

En els últims anys jugant a jocs d'ordinador concretament els que se centren en explotació de recursos, creació de fàbriques, automatització de processos va sorgir el joc Factorio que com s'ha explicat el disseny de fàbriques conté diferents tipus de problemes d'optimització arran d'això vaig descobrir que des de la universitat de Saint Andrews ja s'havia abordat aquest problema usant Essence Prime i com a causa de la tecnologia usada es proposava una millora a futur que tractava de la implementació del model en SMT la qual permet usar nombres reals. Aquesta va ser una de les motivacions principals, la possible ampliació i millora d'un model usant una tecnologia nova per mi.

Així doncs, el propòsit del projecte és crear una eina amb interfície gràfica còmoda d'usar per a l'usuari i un optimitzador darrere que sigui capaç de trobar les configuracions òptimes en funció de diferents criteris (maximitzar la producció, minimitzar els elements que s'han d'usar...).

1.3 Objectius del projecte

Els objectius que es van esbossar a l'inici del projecte van ser els següents:

- Agafar pràctica amb l'API del SMT solver Z3, resolent problemes clàssics com les N-reines.
- Implementar un model bàsic en SMT fent ús de les seves característiques envers SAT solvers (ús de nombres reals, dominis no definits).
- Crear una interfície gràfica que permeti la generació d'instàncies de manera còmoda i generar un conjunt d'instàncies que posin a prova el model.
- Amb el model bàsic en funcionament, fer una anàlisi de rendiment amb les instàncies generades, realitzar optimitzacions i evolucionar el model.
- Si el rendiment és prou bo, ampliar el model amb més mecàniques del joc.
- Crear un model que usi instàncies resoltes com a elements de fabricació per crear dissenys subòptims més grans.
- Crear una interfície gràfica amb la qual poder visualitzar les instàncies amb una bona presentació.

Tenint en compte que el model s'ha hagut de crear des de zero a causa del canvi de tecnologia hi ha un parell d'objectius que eren massa ambiciosos i no s'han pogut assolir.

Tot i que el rendiment del model ho permet no s'han incorporat més mecàniques (cintes subterrànies, cintes amb múltiples objectes...), ja que la seva complexitat implicava molts canvis i hores de desenvolupament que haurien compromès la finalització del treball.

A banda tampoc s'han pogut usar les instàncies resoltes com a objecte de fabricació per al desenvolupament de dissenys subòptims a nivell macro, aquest objectiu, però té molt potencial per a feina futura, ja que no implica fer canvis al model, sinó dissenyar-ne un altre que sigui capaç d'usar els dissenys trobats com a elements de fabricació.

Capítol 2

Estudi de viabilitat

En aquest capítol s'explicarà quins motius van fer que el projecte fos viable i els recursos emprats pel seu desenvolupament.

2.1 Viabilitat del projecte

Els motius que han fet que aquest projecte fos viable han sigut principalment que en tractar-se d'un projecte enfocat més en la recerca no ha calgut fer un pressupost ni hi ha hagut cap despesa. Pel que fa a les llibreries que s'han usat són d'ús lliure i no requereixen llicència.

Per poder fer proves amb el model des del grup de recerca en lògica i intel·ligència artificial es va posar en disposició el clúster per si es volien resoldre instàncies més difícils, tot i que la majoria de proves s'han pogut fer des del meu ordinador personal.

El fet que des de la Universitat de Saint Andrews ja hagués explorat aquest problema donava més seguretat a l'hora dels possibles resultats que es podien esperar, donant així una mica de seguretat, ja que el model bàsic se sabia que era codificable.

Finalment, com la llibreria Z3 que incorpora el SMT solver és de les que més reconeixement té i està molt consolidada amb una API fàcil d'usar del Python, va aportar la seguretat que connectar la interfície gràfica no seria gens difícil gràcies al fet que Python és dels llenguatges de programació més usats en l'actualitat i permet usar infinitat de llibreries tant si es volia desenvolupar la interfície d'usuari des del mateix Python com si es volia fer Web i després comunicar-ho usant una infraestructura client-servidor.

Una de les coses que no es va tenir en compte a l'inici del projecte era les implicacions legals que pot tenir desenvolupar un projecte relacionat amb un producte de pagament en aquest cas el joc Factorio.

Segons els termes de servei que s'expliquen a la pàgina web oficial del joc, l'única implicació es deu a l'ús dels Sprites del joc a la interfície web que s'ha desenvolupat sobre els quals l'estudi Wube Software Ltd. té tots els drets i si ho consideressin podrien demanar-me que les treies. Però com es tracta d'un treball de recerca, es fa

menció que els drets de les imatges pertanyen a l'estudi Wube Software Ltd. i no es pretén monetitzar de cap manera el projecte desenvolupat no hi hauria d'haver cap problema.

A banda aquest no és el primer projecte sobre el joc que usa Sprites oficials, existeix l'eina "Factorio-SAT" que se centra en el balanceig de flux i va rebre bastant suport per la comunitat del joc i no hi ha cap acció des de Wube Software Ltd. per retirar els Sprites.

2.2 Recursos usats

Pel desenvolupament del model i les proves que se li han fet, s'ha necessitat un IDE per redactar el codi, concretament s'ha usat PyCharm desenvolupat per JetBrains del qual tenim una llicència proporcionada a tots els alumnes des de la UdG.

Un ordinador per poder executar totes les instàncies, les especificacions de l'ordinador són: processador AMD Ryzen 5700U de 8 cores, 16 gigabytes DDR4 de memòria RAM.

Les llibreries usades són Z3 que com ja s'ha comentat incorpora el SMT solver usant per a l'optimització de les fàbriques i Flask una llibreria de Python que permet iniciar un servidor, crear endpoints i és el que ha permès poder fer la interfície gràfica en HTML i JavaScript, l'explicació en detall de les llibreries es pot trobar al capítol 6 de Requisits del Sistema.

Capítol 3

Metodologia

Com s'ha esmentat anteriorment aquest projecte s'enfoca principalment en l'àmbit de la recerca i no en el desenvolupament i manteniment d'un producte així que no s'ha usat cap metodologia de treball estandarditzada com per exemple SCRUM.

El que s'ha fet ha sigut fer reunions quinzenals amb el tutor on s'ha exposat la feina feta els últims 15 dies, parlat si calia modificar alguna part de la feina feta i s'ha plantejat la feina que s'havia de fer el següent Sprint de quinze dies, que principalment ha sigut implementar les restriccions que codifiquen cadascuna de les mecàniques del joc, fer una petita documentació per poder posar en context el funcionament al tutor i dur a terme petits testos sobre les restriccions implementades per comprovar-ne el funcionament.

Aquesta principalment ha sigut la metodologia tot i que algunes setmanes fos per temes personals, perquè alguna de les tasques del Sprint es compliqués o bé perquè em devies desenvolupar la interfície web, alguns Sprints s'han hagut d'allargar tres setmanes o per algun Sprint no s'ha pogut desenvolupar tot el que s'havia plantejat. Aquesta metodologia dels Sprints s'ha seguit fins al final del projecte passant per tots els objectius principals plantejat a l'inici del projecte: desenvolupament del model bàsic, aplicació de millors i optimitzacions, desenvolupament de la interfície web i redacció de la memòria.

Capítol 4

Planificació

4.1 Planificació inicial del projecte

A les primeres reunions realitzades amb el tutor es van esbossar quins haurien de ser els passos que s'haurien de seguir per poder complir amb els objectius que plantejats, així i tot, és molt difícil planificar exactament tota la feina que s'ha de realitzar, ja que és difícil estimar el temps que es tarda a fer cada tasca ja sigui per limitacions de la tecnologia usada que poden causar que certes modelitzacions siguin inviables.

La planificació inicial susceptible a canvis és la següent:

- Adquisició de coneixement sobre l'API de Z3.
- Entendre el treball fet per en Sean Patterson veure els seus viewpoints i decidir si se'n podia aprofitar algun.
- Implementar mecànica a mecànica del joc al model i avaluar el seu funcionament.
- Crear un set d'instàncies per posar a prova el model bàsic.
- Analitzar el model actual i fer iteracions amb millors tot analitzant el seu rendiment.
- Donat el rendiment valorar si és possible afegir més complexitat al model.
- Crear una interfície gràfica per visualitzar les instàncies resoltres.

Durant el transcurs del desenvolupament del projecte s'ha seguit la planificació que s'havia pensat a l'inici, però per millor descripció de les tasques, tot seguit què s'ha fet concretament a cada fase i les petites desviacions de la planificació inicial que han sorgit.

4.2 Seguiment de la planificació

4.2.1 Ús de l'API de Z3

A tots els projectes que requereixen l'ús d'una nova tecnologia hi ha una primera fase d'estudi i exercicis. En aquest cas per agafar experiència amb l'ús de l'API de

Z3 s'ha seguit la guia Z3 Tutorial [2] publicada a Google Collaborate, en aquest document es proposen tot d'exercicis, explicacions dels diferents tipus de variables, operadors lògics i funcions que ofereix Z3. Alguns d'aquests exercicis tracten de problemes clàssics com el de les N-reines, Send More Money d'entre altres.

Aquesta guia juntament amb la lectura complementària dels punts més importants de la documentació Programming Z3 [3] creada pels contribuïdors del Theorem Prover Z3 des de l'equip de recerca de Microsoft, ha sigut ser essencial per aprendre i assolir una base sòlida de coneixement sobre Z3 per poder començar a plantejar viewpoints i abordar el problema principal.

4.2.2 Anàlisi del treball fet a St. Andrews

Per aquesta fase s'ha fet ús d'un Notebook de Jupyter per poder fer petits experiments sobre els viewpoints explicats al treball Towards Automatic Design of Factorio Blueprints [4]. El model creat al paper mencionat s'usa una arquitectura multinivell on se secciona el problema en diferents fases, aquesta decisió es veu influenciada per la limitació de PDDL de modelar variables contínues i a les millores del model proposen unificar el model per ajudar al solver a fer propagació i no comprovar solucions redundants.

Per aquest motiu la majoria del plantejament realitzat al paper no s'ha pogut usar. En un dels nivells de l'arquitectura que es comenta al paper, s'explica la representació que es fa per modelar les rutes que les cintes de transport segueixen, concretament es parla de la representació incremental i direccional, les quals s'han usat al model desenvolupat en aquest projecte. Aquesta representació s'ha implementat de manera temporal al Notebook mencionat per analitzar el rendiment i escalabilitat de la modelització.

Finalment, del paper també s'han extret algunes les mecàniques principals (rutes, tipus d'objectes i quantitat d'objectes) que conformen part del model del projecte.

4.2.3 Implementació del model bàsic

El modelatge i implementació de les restriccions ha estat la que més temps ha dut completar, ja que moltes de les modelitzacions que codifiquen les mecàniques bàsiques del joc han requerit ser modificades múltiples vegades. Concretament, una de les parts més importants del model (quantitat d'objectes), ha passat per tres iteracions fins que el seu comportament ha sigut el desitjat.

En aquesta fase s'han implementat totes les restriccions necessàries per crear un model base completament funcional sobre el qual, retocar alguns comportaments, implementar optimitzacions i fer una anàlisi del rendiment per veure si el projecte avança per bon camí i si realment s'està fent una aportació valiosa al camp de la lògica i la intel·ligència artificial.

4.2.4 Creació d'instàncies

Amb la base del model implementada cal un set d'instàncies per poder veure el comportament del model, fer proves, descobrir si hi ha algun error en la implementació i comprovar el rendiment.

Inicialment, es volien crear les instàncies manualment, buscant les receptes que les fàbriques poden usar, seleccionant les entrades i sortides dels materials, els tipus... aquesta manera és molt ineficient a causa de l'enorme quantitat de receptes que el joc conté, com aquestes receptes poden requerir altres receptes, els materials que

aquestes impliquen i com és de poc intuïtiu introduir coordenades d'una graella sense poder tenir una representació gràfica per ubicar on està cada casella. Així que aquesta fase de la planificació no només s'han generat totes les instàncies necessàries sinó que també s'ha hagut de desenvolupar una eina per poder-les generar de manera automàtica a partir d'una interfície gràfica fàcil d'usar.

Les instàncies estan enfocades a posar a prova diferents aspectes del model, ja sigui perquè requereixen moltes receptes, l'espai és reduït o les receptes requereixen molts o poc materials d'entrada. També s'ha jugat amb els ràtios d'entrada i sortida de les receptes.

4.2.5 Aplicar millors sobre el model base

Després d'analitzar el model base s'ha fet una reunió amb el tutor on ha sorgit una optimització que simplifica força el model. Durant la implementació de l'optimització també han sorgit un upper i lower bound que han ajudat força. Finalment, hi ha hagut 3 canvis respecte al comportament del model per replicar amb millor precisió el funcionament real del joc.

Amb aquestes millors aplicades, s'ha fet un script en Python que conté el set d'instàncies usat per fer proves sobre el rendiment i s'ha deixat l'ordinador resoldre-les amb un timeout de 1800 segons. Tenint en compte que hi ha tres millors substancials i s'han fet optimitzacions seguint tres criteris diferents, el procés de resoldre instàncies ha durat unes tres setmanes.

4.2.6 Visualització de les instàncies resoltres

Queda fer-ho

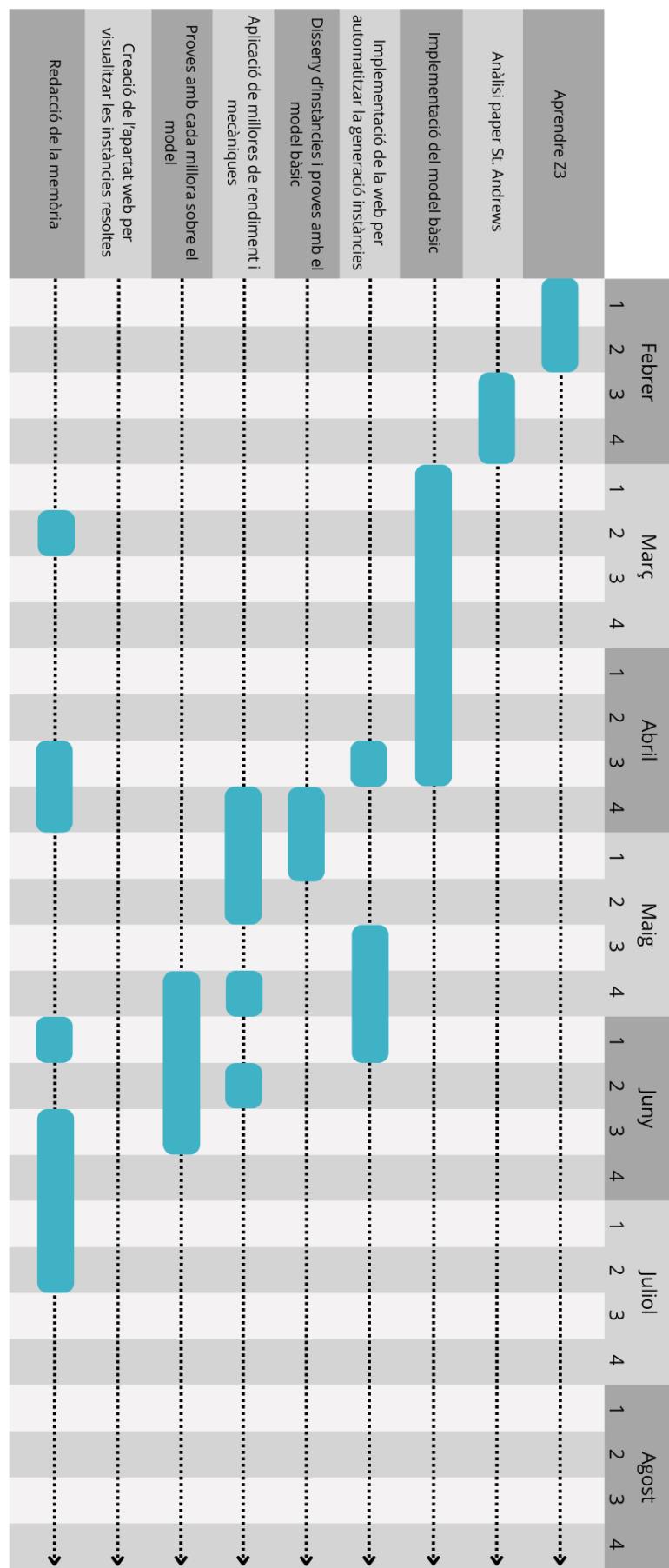


FIGURA 4.1: Diagrama de Grantt del desenvolupament del projecte

Capítol 5

Marc de treball i conceptes previs

Aquí s'expliquen tots els conceptes que cal saber per poder entendre tot el que s'ha desenvolupat al projecte, des de la modelització fins al disseny web i la connexió client-servidor.

5.1 Problemes de satisfacció de restriccions

Els problemes de satisfacció de restriccions en anglès CSP, es defineixen com a un conjunt de n variables $X = \{X_1, X_2, \dots, X_n\}$, un conjunt $D = \{D_1, D_2, \dots, D_n\}$ de dominis per cada variable i un conjunt $R = \{R_1, R_2, \dots, R_n\}$ de restriccions aplicades sobre les variables del conjunt X , aquestes restriccions són relacions k àries entre les variables on l'objectiu és trobar una assignació sobre el conjunt de variables X fent ús del domini D de cada variable que satisfaci totes les restriccions del conjunt R . Les assignacions de valors a les variables s'anomena interpretació, aquesta interpretació no té per què satisfer les restriccions imposades sobre les variables del conjunt X , però la interpretació que si satisfa totes les restriccions s'anomena model. Tot seguit un exemple del famós problema send more money:

Suposem que tenim les variables $X = \{S, E, N, D, M, O, R, Y\}$ on el domini de cada variable és $D = \{[0..9], \dots, [0..9]\}$ i el que volem és satisfer la restricció $R_1 = \{1000S + 100E + 10N + D + 1000M + 100O + 10R + E = 10000M + 1000O + 100N + 10E + Y\}$ a més també volem que tots els valors que prenguin les variables siguin diferents entre ells $R_2 = \{S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y\}$.

Una possible interpretació podria ser $S = 0, E = 1, N = 2, D = 3, M = 4, O = 5, R = 6, Y = 7$ l'avaluació d'aquesta assignació sobre les restriccions seria:

$$R_1 = \{4684 = 45217\}$$

$$R_2 = \{0 \neq 1 \neq 2 \neq 3 \neq 4 \neq 5 \neq 6 \neq 7\}$$

Aquesta interpretació no és model, ja que la restricció R_1 no se satisfà.

Una interpretació que sí que és model és la següent: $S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2$

Ja que satisfa les dues restriccions:

$$R_1 = \{10652 = 10652\}$$

$$R_2 = \{9 \neq 5 \neq 6 \neq 7 \neq 1 \neq 0 \neq 8 \neq 2\}$$

Amb aquest exemple es pot veure com aquest tipus de problemes no tenen un "mètode" per ser resolts sinó que requereixen prova i error fins que no es topa amb la

solució. La complexitat d'aquest tipus de problemes s'estudia a branca de la complexitat computacional tot seguit s'expliquen les idees bàsiques sobre aquest camp de la computació.

5.2 Complexitat computacional

Per poder definir les diferents complexitats computacionals dels problemes de CSP, cal introduir el concepte de màquina de Turing la qual serveix de base per establir que és computable i que no.

5.2.1 Màquina de Turing

La màquina de Turing, inventada per Alan Turing el 1936, és un model de còmput que descriu la màquina més simple capaç de poder executar qualsevol algorisme informàtic. La màquina treballa amb un alfabet finit i consta d'una cinta de longitud infinita dividida en cel·les, cada cel·la pot contenir qualsevol símbol de l'alfabet. La màquina també compta amb un capçal el qual apunta a la cel·la actual i és capaç de fer operacions de lectura i escriptura sobre la cel·la que apunta. A més la màquina compta amb un conjunt finit d'estats i una taula finita d'instruccions, en funció del símbol llegit i l'estat actual, s'escull una de les instruccions de la taula que poden ser, moure el capçal a la dreta o esquerra, esborrar o escriure un símbol a la cel·la actual o bé canviar d'estat.

5.2.2 Classes de complexitat

A la teoria de la computació hi ha moltes classes de complexitat, referents a l'espai en memòria necessari per resoldre un problema i el cost en temps requerit. Per aquest treball només s'explicaran les classes que fan referència al temps de còmput, ja que és el factor principal que afecta el problema de CSP.

Classe P

El conjunt de complexitat P conté tots aquells problemes que tenen un algorisme específic per ser resolts, on el cost en temps d'aquest algorisme augmenta polinomialment en funció del nombre de variables implicades al problema. Per exemple trobar el valor mínim en una llista de nombres té un cost lineal $O(n)$ que augmenta en funció de la longitud de la llista, l'ordenació dels nombres d'una llista és un altre problema de la classe P que pot ser resolt amb cost $O(n \log n)$. També es diu que un problema és de complexitat P si una màquina de Turing determinista els pot resoldre en temps polinòmic.

Classe NP i NP-complet

Els problemes de la classe NP (Nondeterministic Polynomial Time), són aquells que poden ser resolts en temps polinòmic per una màquina de Turing no determinista, és a dir una màquina de Turing la qual de manera no determinista sap a quins estats ha de canviar per trobar la solució en temps polinòmic. L'exemple més conegut d'aquest tipus de problema tracta del problema de satisfacció (SAT) on donat un conjunt de variables booleanes i una CNF és volt trobar quina assignació fa que la CNF avalui cert. A més SAT forma part de la classe NP-complet que implica que qualsevol problema NP pot ser reduït en temps polinòmic a SAT. Finalment, els

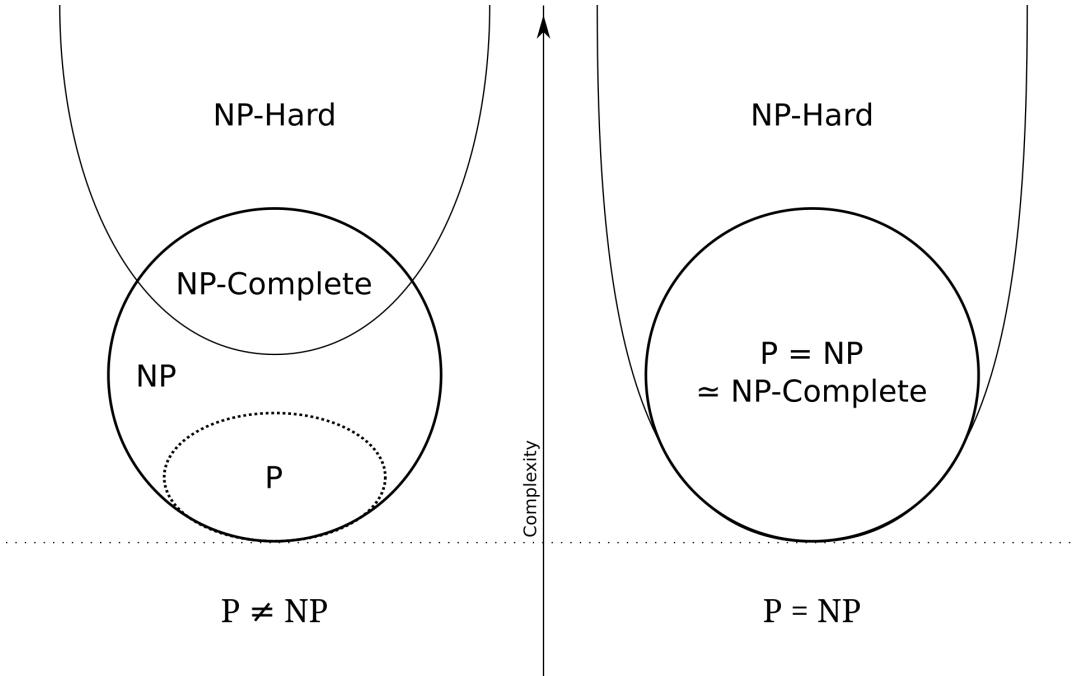


FIGURA 5.1: Comparació de classes de complexitat si $P \neq NP$ i $P = NP$

problemes NP tenen la particularitat que una solució trobada pot ser verificada en temps polinòmic, per exemple per al problema SAT seria tan senzill com assegurar que cada clàusula evalua cert el qual tindria un cost lineal $O(n)$.

NP-hard

La classe NP-hard és de les més curioses, ja que la seva intersecció amb el conjunt NP forma el conjunt de la classe NP-complet, es poden reduir en temps polinòmic a NP-hard, però no tots els problemes NP-hard es poden reduir en temps polinòmic a problemes NP, a més, i molt important, els problemes NP-hard que no formen part del conjunt NP no es poden verificar en temps polinòmic per una màquina de Turing determinista. Per exemple alguns problemes d'optimització són NP-hard, ja que donada una solució comprovar que sigui l'òptima requereix saber totes les solucions per determinar si la solució trobada efectivament és l'òptima.

Totes aquestes definicions són certes sota l'assumpció que $P \neq NP$ i fins que es pugui demostrar el contrari les classes de complexitat continuaran sent així. En cas que es pogués demostrar $P = NP$, qüestió que tracta d'un dels set problemes del mil·lenni la representació de les classes de complexitat tenint en compte els dos escenaris pot ser vista a la figura 5.1.

5.3 SAT

Com s'ha explicat a la secció 5.2.2, SAT es troba dins el conjunt NP-complet, aquesta característica sumada a la simplicitat del problema, ha causat que sigui usat com a base per solucionar problemes del conjunt NP a causa de la propietat que tot problema del conjunt NP pot ser reduït a NP-complet. La tecnologia usada tracta dels SAT solvers els quals usen tècniques molt avançades per reduir el màxim el cost de la cerca per força bruta implícita als problemes d'aquesta complexitat.

Exemple del problema SAT. Suposem que tenim la següent forma normal conjuntiva (CNF):

$$(a \vee b \vee c) \wedge (\neg c \vee \neg b) \wedge (\neg a \vee \neg b) \wedge (\neg a \vee \neg c)$$

Una interpretació possible seria:

$$[a = \text{cert}, b = \text{false}, c = \text{false}]$$

$$(\text{cert} \vee \text{false} \vee \text{false}) \wedge (\neg \text{false} \vee \neg \text{false}) \wedge (\neg \text{cert} \vee \neg \text{false}) \wedge (\neg \text{cert} \vee \neg \text{false})$$

$$\text{cert} \wedge (\text{cert} \vee \text{cert}) \wedge (\text{fals} \vee \text{cert}) \wedge (\text{fals} \vee \text{cert})$$

$$\text{cert} \wedge \text{cert} \wedge \text{cert} \wedge \text{cert}$$

cert

Aquesta satisfà la fórmula, és model. A més aquesta CNF té la propietat que les assignacions que satisfan la fórmula són aquelles on només un literal és cert.

a	b	c	$(a \vee b \vee c) \wedge (\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee \neg c)$
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	0

Un dels problemes de SAT és que com que tracta de la satisfacció de les clàusules d'una CNF les quals estan formades per literals que són variables booleanes, hi ha problemes sobretot si aquests tracten amb variables numèriques que són molt difícils de reduir a SAT i encara que siguin reduïbles en certs casos generen tantes clàusules que usar un SAT solver passa a ser una opció inviable. Hi ha diverses alternatives que permeten fer modelitzacions similars a les que es farien amb SAT, però amb l'opció de poder usar variables de diferents tipus, per aquest projecte s'ha usat SMT (SAT Modulo Theories).

5.4 SMT

SMT o SAT Modulo Theories, tracta del problema de determinar si una fórmula matemàtica és satisfacible. A diferència del problema SAT, SMT incorpora múltiples teories, des d'aritmètica lineal fins a comparacions lògiques entre funcions no interpretades. Aquestes teories permeten l'ús de diferents tipus de dades (enters, reals, strings, llistes, arrays de bits...). A més també permet crear fórmules lògiques igual que a SAT però amb l'ús de les teories.

Els SAT solvers tenen un nucli que està especialitzat a trobar les assignacions satisfacibles de la fórmula fent ús de tècniques com CDCL. SMT aprofita aquest nucli i en fa una abstracció incorporant l'ús de les teories, tot seguit un exemple per entendre millor l'abstracció. Suposem que tenim la següent fórmula:

$$(c \leq 5) \wedge (a \geq 3) \wedge (5a + b = c - 3)$$

El primer pas és identificar cada teoria present a la fórmula i abstreure'n una variable booleana perquè el nucli SAT pugui resoldre.

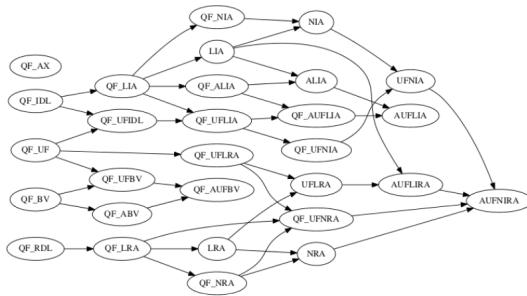


FIGURA 5.2: Teories suportades per SMT-lib [5]

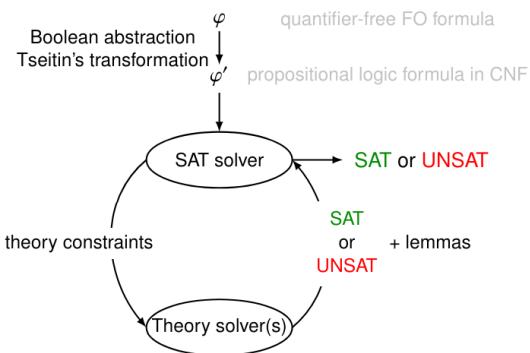


FIGURA 5.3: Esquema d'un SMT solver lazy [5]

$$[A = (c \leq 5), B = (a \geq 3), C = (5a + b = c - 3)] \\ A \wedge B \wedge C$$

Dins de les teories que la llibreria de SMT-lib suporta 5.2, les variables A i B tracten de la teoria (integer rational difference logic o QF_IDL) i la variable C la teoria (real/integer linear arithmetic). Amb l'abstracció feta i les teories identificades el solver primer resol la part SAT de la fórmula, si es troba una assignació vàlida es criden els nuclis que resolen les teories, en cas que hi hagi una assignació que satisfaci les variables abстretes la fórmula és satisfacible, en cas contrari, el nucli del SAT solver aprèn com a lema que la variable no pot ser assignada un valor positiu i continua fent solving fins que es demostra satisfacibilitat o no hi ha més assignacions disponibles.

En aquest exemple com la fórmula és molt simple el SAT solver demana als nuclis de les teories esmentades que avaluïn cert i aquestes per exemple podrien respondre el següent:

$$A \Rightarrow c = 4 \Rightarrow 4 \leq 5 \Rightarrow cert$$

$$B \Rightarrow a = 3 \Rightarrow 3 \geq 5 \Rightarrow cert$$

$$C \Rightarrow b = -14 \Rightarrow (15 - 14 = 4 - 3) \Rightarrow cert$$

L'esquema de la figura 5.3 representa el procés de solving d'un SMT solver usant lazy solving.

5.5 Elements bàsics del joc Factorio

Com s'ha explicat prèviament, Factorio és un joc d'automatització i ampliació de fàbriques en un món 2D basat en una graella. Els elements que el joc posa a disposició

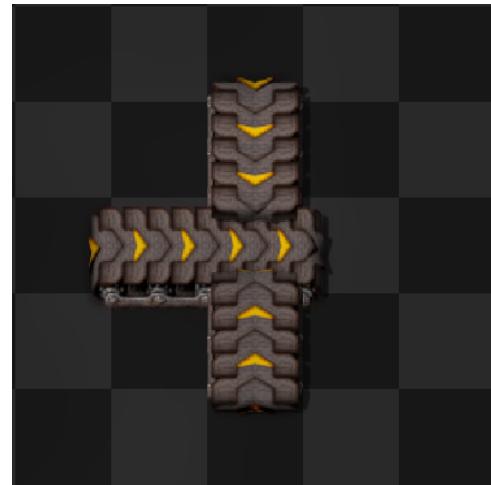
del jugador en són molts, així i tot, els bàsics per poder construir i automatitzar la producció d'objectes en són tres: el *inserters*, *conveyor belts* i els *assemblers* tot seguit una explicació en detall del comportament de cadascun.

5.5.1 Cinta transportadora

Les cintes transportadores o *conveyor belts* serveixen per transportar elements d'un punt a un altre. Aquestes ocupen 1x1 caselles a la graella del món i poden prendre qualsevol de les 4 direccions cardinals. Transporten objectes de la casella actual a la següent casella apuntada per la cinta. Aquestes poden rebre objectes per qualsevol de les tres caselles adjacents diferents de la casella de sortida (la que apunta la cinta). Al joc les cintes consten de dues pistes cadascuna amb la mateixa capacitat de transport d'objectes, però no s'ha tingut en compte aquesta mecànica, ja que de la manera que funciona afegeix molta complexitat. La capacitat de transport d'una cinta, tenint només en compte una de les pistes, és de 450 objectes per minut.



((A)) Conveyor belt a la graella del joc



((B)) Entrades permeses a una conveyor belt

FIGURA 5.4: *Conveyor belt* dins el joc i les seves possibles entrades

5.5.2 Inseridor

Els inseridors o *inserters* són braços robòtics que ocupen 1x1 caselles a la graella del món, permeten inserir i treure elements de cintes i *assemblers*.

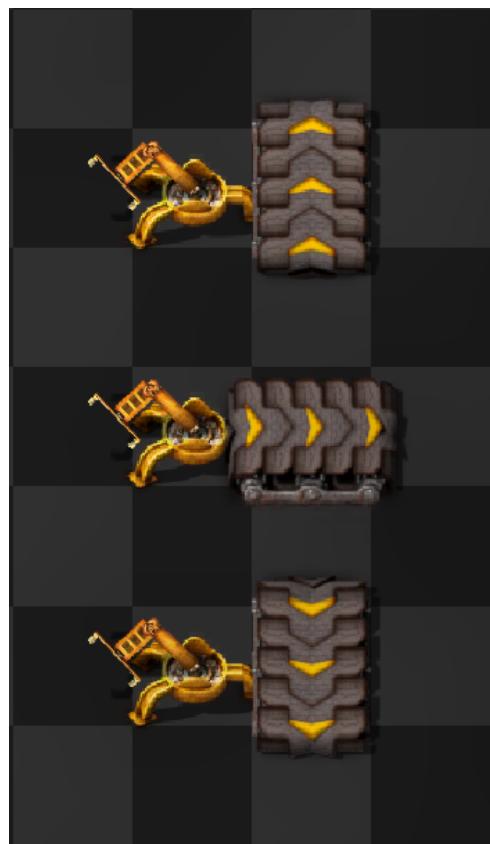
Aquests poden treure elements de cintes que apuntin en qualsevol direcció i inserir objectes a cintes que no apuntin cap a ell. L'inseridor és l'únic element que pot suprir d'objectes a un *assembler* i té el càncer que la seva capacitat de transport és de 50 objectes per minut.



FIGURA 5.5: Inserter a la graella del joc



((A)) Entrades permeses



((B)) Sortides permeses

FIGURA 5.6: Orientacions de les cintes d'on un *inserter* pot treure i agafar objectes

5.5.3 Assemblador

Els assembladors o *assemblers* ocupen un espai de 3x3 caselles a la graella del món, aquests només poden rebre i treure objectes a través d'*inserters* que estiguin en el mateix eix que qualsevol de les 12 caselles adjacents a l'*assembler*.

La funcionalitat dels *assemblers* és convertir els materials d'entrada en objectes refinats, això es fa mitjançant receptes.



((A)) Assembler a la graella del joc

((B)) Inserters interactuant amb l'assembler

FIGURA 5.7: Representació del *assembler* dins el joc

Receptes

Les receptes només poden estar associades als *assemblers*, aquestes indiquen quins i quants objectes són necessaris per a la fabricació d'un altre objecte en un temps determinat.

Amb el temps que tarda un *assembler* a produir els objectes d'una recepta i la quantitat d'objectes que requereix, podem calcular el nombre d'objectes per minut d'entrada i sortida màximes d'una recepta. Això ens serà molt útil més endavant per modelar receptes i el flux d'objectes.

En cas que un *assembler* rebi més objectes per minut dels requerits per la recepta, aquests s'acumularan als *inserters* i cintes que transporten l'objecte al *assembler*, d'altra banda, si l'*assembler* rep menys objectes per minut del que la recepta requereix, l'*assembler* haurà d'esperar a tenir els objectes per iniciar la recepta fent que velocitat dels objectes fabricats es redueixi.

Totes les receptes que es produeixen al *assembler* sempre generen un sol tipus d'objecte de sortida.

FIGURA 5.8: Informació de la recepta d'un *assembler*

5.6 Conceptes relacionats amb els elements bàsics del joc

Amb elements bàsics que poden constituir un *blueprint* ja explicats, queda definir com aquests elements han d'interactuar entre ells per formar una fàbrica que respecti les mecàniques del joc, així doncs tot seguit s'explica com s'han dividit les interaccions entre elements, objectes que es produeixen a la fàbrica...

5.6.1 Ruta

Una de les parts més importants del model és definir com la concatenació d'*inserters* i *conveyor belts* conformen una ruta. Perquè una ruta sigui vàlida no pot crear cicles, és a dir que una cinta o *inserter* no pot entrar objectes a una cinta la qual era anterior a ella mateixa. També s'ha de tenir en compte l'orientació dels *inserters* i cintes per complir amb les entrades i sortides vàlides anteriorment descrites. Una ruta també es pot bifurcar i unir.

Els inicis de ruta poden ser les caselles marcades com a entrada o bé la sortida d'un objecte d'un *assembler*. D'altra banda, els finals de ruta poden ser les caselles marcades com a sortida o bé l'*inserter* que afegeix elements a un *assembler*.

5.6.2 Tipus d'objectes

Les cintes i *inserters* han de poder dur objectes per les rutes que conformen, aquí entra la noció del tipus d'objectes, que depenen dels d'objectes específicats a les caselles d'entrada i sortida del *blueprint*, juntament amb els objectes que un *assembler* genera en funció de la recepta que tingui associada. Cal assegurar que a les 12 caselles adjacents a un *assembler* les quals tenen *inserters* entrant o sortint, no duguin o treguin objectes que la recepta associada al *assembler* demana, també cal assegurar que una casella que forma part d'una ruta només pot dur un tipus d'objecte i finalment cal assegurar la correcta propagació dels objectes per les rutes.

5.6.3 Quantitat d'objectes

A banda de portar objectes, les rutes poden estar més o menys carregades, és a dir que cal alguna manera de saber quants objectes per minut porta una cinta o *inserter* per poder saber quants objectes per minut ha de produir la recepta associada a un *assembler*.

La recepta associada a un *assembler*, com bé ha quedat explcitat anteriorment, requereix un cert nombre d'objectes per minut per produir-ne uns altres a una densitat específica. Un dels problemes principals és que un *assembler* pot rebre objectes requerits per la recepta en diferents quantitats que poden ser superiors o inferiors al *ratio* que marca la recepta, però els objectes que entrin en major quantitat no ho podrán fer durant gaire temps, ja que ràpidament l'*assembler* se saturarà d'objectes fent que la quantitat d'entrada s'anivelli a la quantitat respectiva de l'objecte que crea el coll d'ampolla.

Determinar correctament la quantitat d'objectes que hi ha a cada casella del *blueprint* és una de les parts més crucials del problema, ja que és la que ens permetrà saber quants elements s'estan produint, cosa que volem maximitzar.

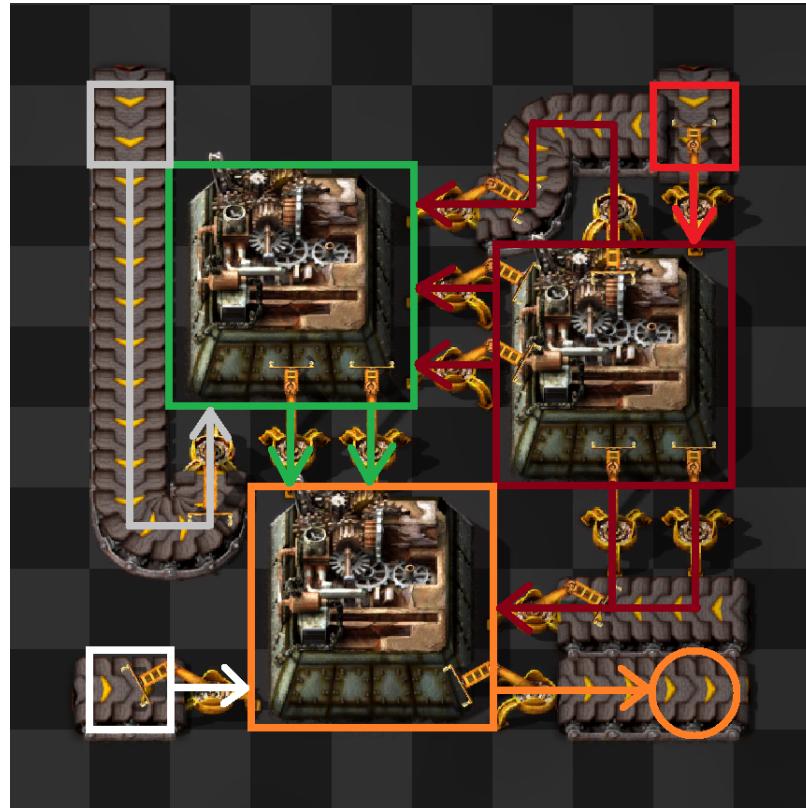


FIGURA 5.9: Exemple base

5.7 El problema del blueprint

El problema que es vol resoldre s'anomena el problema del *blueprint*, el qual donades les entrades i sortides dels objectes, l'objecte que es vol produir i l'espai del qual es disposa, s'ha de maximitzar la quantitat produïda de l'objecte de sortida.

Per entendre millor apartats futurs on es parla d'implementacions concretes, es presenta un exemple, el qual tracta d'una instància resolta pel model. Amb aquest exemple es posa en context el problema i s'explica perquè aquest no és gens senzill.

5.7.1 Exemple del problema

Els inputs d'aquesta instància són les següents:

- Mida: 8x8
- Entrades: (0,0) "quadrat gris" Xapes de ferro, (0,7) "quadrat vermell" Xapes de coure, (7,0) "quadrat blanc" Barres de plàstic
- Sortides: (7,7) "cercle taronja" Circuits avançats
- Objecte a produir: Circuits avançats

Com es pot veure a l'exemple resolt 5.9, per produït Circuits avançats s'han necessitat tres *assemblers*, un per cada recepta implicada en la producció de Circuits avançats, concretament es requereixen les receptes:

Recepta	Requereix	Produeix
Circuit avançat	40 Cable de coure 20 Circuits electrònics 20 Barres de plàstic	10 Circuits avançats
Circuit electrònic	360 Cable de coure 120 Plaques de ferro 20 Barres de plàstic	120 Circuits electrònics
Cable de coure	120 Plaques de coure	240 Cables de coure

TAULA 5.1: Materials requerits i produïts per cada recepta

- Cable de coure "*assembler* emmarcat en carmí"
- Circuits electrònics "*assembler* emmarcat en verd"
- Circuits avançats "*assembler* emmarcat en taronja"

Com l'espai del *blueprint* és molt ajustat hi ha rutes que només tracten d'un *inserter* que agafa els objectes produïts d'un *assembler* i els insereix directament al *assembler* adjacent.

Pel que fa a la distribució dels objectes primer cal saber la quantitat d'objectes per minut que requereix i produeix cada recepta.

Pel que fa als materials en cru que entren per les caselles emmarcades a la imatge, estan entrant les següents quantitats:

Material	Quantitat d'entrada
Placa de coure	50
Placa de ferro	98.06
Barra de plàstic	20.0138

D'aquestes quantitats d'entrada no s'aprofiten tots els objectes pel fet que alguns *assemblers* no tenen prou *inserters* de sortida per poder aprofitar tot el material. Aquesta "pèrdua" es tradueix en material que s'acumula a les cintes d'on els *inserters* agafen els materials d'entrada pel *assemblers*, concretament la cinta a la posició (7,0) acumula 0.138 barres de plàstic, fent que la quantitat d'entrada de l'*assembler* de circuits avançats sigui de 20 barres de plàstic per minut. D'altra banda, la cinta a la posició (5,1) acumula 78.06 plaques de ferro, fent que l'entrada a l'*assembler* sigui de 20 plaques de ferro per minut. Finalment, les plaques de coure que entren per la casella (0,7) s'aprofiten totes.

Ara que ja s'han explicat les quantitats d'entrada i els materials que cada recepta requereix, es pot veure com l'*assembler* carmí per poder processar les 50 plaques de coure per minut ha de poder extreure 100 cables de coure això ho fa mitjançant 5 *inserters*, els quals es divideixen la quantitat de sortida en parts proporcionals fent que 3/5 parts, és a dir 60 cables, es destinin a la producció de circuits electrònics i 2/5 parts, és a dir 40 cables, es destinin a la producció de circuits avançats.

Els cables destinats a la producció de circuits electrònics s'aprofiten tots, ja que l'entrada de plaques de ferro és de 20/min i com s'ha vist a la taula 5.1 per cada placa

de ferro es necessiten tres cables de coure, així que per 20 plaques de ferro s'entren 60 cables de coure, produint 20 circuits electrònics per minut. Aquests 20 circuits electrònics s'aprofiten tots per la recepta de circuits avançats juntament amb les 20 barres de plàstic d'entrada i la resta de cables de coure, fent que l'*assembler* que produeix circuits electrònics funcioni al 100% de rendiment, fent que si es volguessin produir més circuits avançats es requerís un segon *assembler* el qual per les dimensions del *blueprint* no sigui viable.

Amb aquest exemple es pot entendre més la complexitat del problema i la quantitat de factors que hi juguen. A més també serveix per ser usat com a ajuda visual per entendre implementacions específiques en apartats posteriors.

Capítol 6

Requisits del sistema

Per aquest projecte s'han usat dues llibreries principals que separen el treball en dues parts. Front end, es tracta de la web on es poden generar i visualitzar instàncies, back end fet en Python que conté tota la lògica i les modelitzacions.

Com és un projecte fet en Python els requisits són molt fàcils d'instal·lar en un ordinador sense importar si el sistema operatiu és Windows o Linux. Els passos a seguir són:

Pas 1: Tenir una màquina amb un intèrpret de Python amb la versió 3.9

Pas 2: Crear un entorn virtual amb la comanda `python -m venv <directory>`

Pas 3: Activar l'entorn virtual mitjançant la comanda `venv`

`Scripts`
`activate.bat` des d'un terminal cmd o `source myvenv/bin/activate`
 des d'un sistema Linux o MacOS.

Pas 4: Clonar el repositori en un directori nou dins el directori on s'ha creat l'entorn virtual, l'estrucció de fitxers hauria de ser la següent:

```
projecte/
    venv/                      # entorn virtual
        FactorioPlanner/       # repositori clonat
            script.py
            requirements.txt
            ...

```

Pas 5: Situar-se al directori principal del projecte i executar la comanda `pip install -r requisits.txt`, aquesta comanda instal·larà totes les dependències del projecte a l'entorn virtual.

Pas 6: Executar la comanda `python mainWeb.py` per iniciar el servidor.

Pas 7: Connectar-se al servidor localment des de qualsevol navegador usant l'URL `http://127.0.0.1:5000/`.

A continuació es fa una explicació més en detall de les llibreries principals que formen part dels requisits:

6.1 Z3

Z3 com s'ha explicat per sobre a l'anterior apartat tracta d'un llenguatge de modelització de constraints. La peculiaritat del Z3 és que tracta d'un SMT (Satisfiability Modulo Theories) és a dir generalitza el problema SAT, afegint diferents teories (funcions no interpretades, àlgebra lineal, arrays...) al problema de satisfacció.

Per aquest treball s'ha usat l'API de Z3 des de Python, a causa que al llarg de l'informe s'exposaran múltiples talls de codi, tot seguit es fa una explicació de com s'usen els operadors lògics i defineixen variables Z3 des de Python.

6.1.1 Tipus i declaració de variables

Z3 té molts tipus de variables (reals, enters, arrays, bit vectors, booleanes, uninterpreted functions i tipus definits customizables), per declarar aquestes variables és molt senzill simplement cal usar el constructor donat pel Z3. Una funció molt interessant és que en usar Python com a llenguatge podem guardar les nostres variables en qualsevol mena d'estructura de dades, cal tenir en compte, que l'accés a aquestes estructures només es pot fer mitjançant variables Python així que si cal accedir a una posició d'una matriu en funció del valor pres per una variable Z3, hauríem d'usar la teoria dels Arrays.

Per últim i molt important, el domini de les variables no es defineix a l'hora de declarar-les, sinó que cal definir restriccions que acotin el domini de la variable, crear un tipus enumerat amb el domini preestablert o usar bit vector amb un nombre determinat de bits. Tot seguit exemples de declaració de diferents tipus de variables.

LISTING 6.1: Declaració de variables

```

1 real = Real("real_variable")
2 enter = Int("integer_variable")
3 bit_vector = BitVec("bit_vector_variable", n_bits)
4 array = Array("array_variable", IntSort(), IntSort())
5 function = Function("function_variable", IntSort(), BoolSort())
6 bool = Bool("boolean_variable")
7 color_type, colors = EnumSort("color", ["blue", "orange", "green", "yellow",
    ", "red"])

```

Alguns detalls importants són que els BitVectors necessiten el nombre de bits com a paràmetre, també que aquests es poden interpretar com a nombres amb signe o sense, per diferenciar-ne el tipus és important usar l'operador correcte a l'hora de definir les restriccions sobre variables del seu tipus, més endavant s'ensenyen les diferències.

Les funcions i els Arrays requereixen el tipus de variable amb el qual indexen o prenen com a paràmetre i també el tipus que han de retornar, això s'indica posant "Sort" després del tipus que es vol que prenguin o retornin.

Com s'ha explicat les variables Z3 es poden guardar en estructures Python, aquí un exemple de com es guardarien les files del problema de les n-reines.

LISTING 6.2: Variables Z3 en arrays de Python

```

1 queens = [Int(f"Q_{row + 1}") for row in range(n_queens)]

```

En aquest cas s'usen les llistes per comprensió de Python per declarar un Array amb variables enteres Z3.

6.1.2 Operadors lògics

Per construir fòrmules lògiques necessitem l'ús d'operadors lògics. Des de l'API de Python es proporcionen mètodes per cada operador, alguns d'aquests no són tan visualment entenedors com els d'altres llenguatges centrats en constraint programming com MiniZinc o EssencePrime, així i tot, a continuació es fa una comparació entre els operadors lògics i els de l'API de Z3.

Operador	Z3
$a \wedge b$	<code>And(a, b)</code>
$a \vee b$	<code>Or(a, b)</code>
$\neg a$	<code>Not(a)</code>
$a \oplus b$	<code>Xor(a, b)</code>
$a \Rightarrow b$	<code>Implies(a, b)</code>
$a \Leftrightarrow b$	<code>a == b</code>
$a > b$	<code>a > b, UGT(a, b)</code>
$a < b$	<code>a < b,ULT(a, b)</code>
$a \geq b$	<code>a >= b, UGE(a, b)</code>
$a \leq b$	<code>a <= b, ULE(a, b)</code>
$a \neq b$	<code>a != b</code>

TAULA 6.1: Comparació d'operadors lògics

Com s'ha comentat anteriorment les variables de tipus BitVector requereixen operadors específics si es volen interpretar de manera unsigned" per això els operadors ($>$, $<$, \geq , \leq) tenen les funcions (UGT,ULT,UGE i ULE), per les interpretacions unsigned dels BitVectors.

Finalment, una funcionalitat molt interessant de Z3 és que permet entrar llistes de variables als operadors lògics i aquests l'apliquen per totes les variables de la llista. Per exemple:

LISTING 6.3: Variable Declaration

```

1 int_variable = [Int(f"int_var_{i}") for i in range(10)]
2 less_than_ten = [int_variable[i] < 10 for i in range(10)]
3
4 all_and = And(less_than_ten)
5 all_or = Or(less_than_ten)

```

6.2 Flask

Flask és un framework desenvolupat en Python fàcil d'accendir des de la seva llibreria. El seu objectiu és per proporcionar una interfície de servidor, facilitant mètodes simples per crear endpoints i comunicar de manera senzilla un client, amb el servidor.

En aquest projecte s'ha usat la llibreria per carregar tota la informació necessària de la pàgina web (html, css i fitxers Javascript), accedir a una col·lecció d'imatges per

representar visualment les instàncies i per poder fer peticions al model desenvolupat i rebre els resultats.

Crear endpoints amb Flask és molt senzill tot seguit s'ensenyà com.

6.2.1 Crear Endpoints usant Flask

Per crear un endpoint al nostre servidor és tan senzill com definir una funció estàndard de Python i usar un decorador específic per assignar la ruta.

LISTING 6.4: Declaració d'un endpoint

```
1 from flask import Flask
2 app = Flask(__name__)
3 @app.route("/exemple")
4 def endpoint():
5     print("endpoint d'exemple")
```

Amb aquesta funció creada i el servidor en marxa ja podem fer una petició al servidor usant l'adreça corresponent, si volem que el servidor ens torni informació podem ampliar la informació que afegim al decorador i especificar si es tracta d'un GET o un POST. Per exemple:

LISTING 6.5: Declaració d'un endpoint

```
1 @_app.route('/processar', methods=['POST'])
2 def processar():
3     data = request.get_json()
4     resultat = processar_informacio(data)
5     return jsonify(resultat)
```

D'aquesta manera es rep la informació del client en format JSON, s'extrau i se li aplica el procés que sigui necessari, finalment es passa en format JSON i es retorna al client.

El client pot estar implementat de moltes maneres, per aquest projecte s'ha optat per fer una pàgina web usant JavaScript. L'estruccura de fitxers que ha de tenir el servidor ha de ser la següent.

6.2.2 Estructura de fitxers

Per poder guardar una pàgina web amb tots els elements que comporta (html, css i la lògica en JavaScript), es necessiten dos directoris. Un conté tots els htmls de les planes de la web i una altra contindrà la resta d'arxius que es vulguin guardar al servidor, imatges, fitxers d'estil html, dades en format JSON, scripts per la pàgina web...

A més el codi que llença el servidor s'ha de trobar al directori arrel i la resta de fitxers Python amb la lògica de l'aplicació es poden organitzar com es vulgui.

```
servidor/
    main.py                  # llençador del servidor, endpoints...
    static/                   # arxius, imatges, css, javascript...
        style.css
        weblogic.js
        images/
    templates/               # arxius html per la web
```

```

index.html
application/          # lògica, classes...
    app.py
    logic.py

```

6.2.3 Peticions des del client web

Una de les maneres més estàndard de fer pàgines web és usant JavaScript. Per poder fer peticions des del client al servidor és necessari que es faci de manera asíncrona per no bloquejar la funcionalitat de la pàgina web mentre el servidor rep i processa la petició. Des de JavaScript enviar una petició es fa de la següent manera:

LISTING 6.6: Enviar petició

```

1 function enviarPeticio() {
2     let data = "informacio"
3
4     fetch('/processar', {
5         method: 'POST',
6         headers: {
7             'Content-Type': 'application/json',
8         },
9         body: JSON.stringify(data),
10    })
11    .then(response => response.json())
12    .then(data => {
13        console.log("El servidor ha respost correctament", data)
14    })
15    .catch((error) => {
16        console.error('Error:', error);
17    });
18 }

```

En aquesta funció s'està enviant una petició de tipus POST a l'endpoint /processar, fetch() és una funció asíncrona que rep com a paràmetres l'URL corresponent a l'endpoint del servidor i el contingut que contindrà el paquet enviat.

Un cop enviat el paquet, la resta del codi pot continuar i la funció then() es queda a l'espera que fetch() retorni la resposta del servidor. Si fetch() respon, la primera funció then() captura la informació enviada pel servidor, en cas que hi hagi un problema i el servidor no respongui o no es pugui llegir la informació enviada, catch() llençarà un error informant que la petició ha fallat.

Capítol 7

Estudis i decisions

Fins ara s'han explicat totes les tecnologies usades i les tasques que s'han desenvolupat al projecte, però no s'ha aprofundit en el perquè. En aquest apartat es dona una explicació del perquè de les decisions més importants, des de la selecció de llibreries fins a les diferents modelitzacions que s'han usat al model.

7.1 Llibreria de solving

Com ja s'ha explicat, al projecte s'han usat principalment dues llibreries, la més important tracta de Z3. El motiu principal pel qual s'ha escollit és perquè al paper de St. Andrews el menciona a les propostes de millora. D'aquí s'ha fet una mica de recerca per saber-ne més i s'ha vist que és dels SMT solvers més reconeguts, a més el fet que estigui disponible per quasi tots els llenguatges de programació l'-ha fet un candidat perfecte per desenvolupar el projecte. Per acabar de decidir-la com a llibreria per al projecte s'ha comentat al tutor i en Joan Espasa, membre del grup de recerca de la universitat de St. Andrews i col·laborador al paper mencionat, els quals han confirmat que es podia usar perfectament per desenvolupar el projecte.

7.2 Llenguatge de programació

La llibreria Z3 [6] està disponible per la majoria de llenguatges de programació. Entre ells s'hi troba Python que s'ha decidit usar per a aquest projecte, ja que és dels llenguatges on més ús es fa de la llibreria i on més recursos on-line es poden trobar. A banda el llenguatge en si és fàcil d'usar fent que a l'hora de modelitzar no sigui necessari preocupar-se de les particularitats del llenguatge.

A banda també s'ha escollit amb la idea en ment que en algun moment del projecte s'hauria d'implementar una interfície gràfica i Python té molts recursos per crear-les de manera fàcil.

7.3 Interfície gràfica

Des de l'inici es tenia clar que es volia fer alguna mena d'interfície gràfica per visualitzar les instàncies, ja que els models complexos que contenen moltes variables, interpretar els resultats sense cap mena d'eina que permeti visualitzar els valors presos de les variables es fa difícil.

En un bon principi es volia fer usant alguna llibreria gràfica de les que s'usen per crear videojocs com per exemple PyGame, però a mesura que el projecte ha evolucionat s'ha vist que a banda de crear una eina per poder visualitzar les instàncies resoltes també seria molt favorable crear-ne una altra per automatitzar-ne la generació. Això ha portat a la decisió que la millor opció era fer una web gràcies a la facilitat que té crear una interfície d'usuari amb botons, textos, entrada d'informació, descàrrega de fitxers, imatges, menús...

Arran d'aquesta decisió la creació de la visualització de les instàncies s'ha fet mitjançant l'element canvas que ofereix html, el qual és molt versàtil i fàcil de desenvolupar usant JavaScript.

7.4 Connexió client servidor

Aïllar la interfície gràfica del projecte, crea el problema que ja no es poden comunicar entre ells usant el mateix llenguatge. Per solucionar aquest problema s'ha decidit usar la llibreria Flask que permet connectar la web amb el model que resol les instàncies mitjançant la creació d'un servidor el qual pot rebre les peticions des de la interfície web. Aquesta clara separació de la part visual de la part lògica en un front end i un back end, ha fet que el projecte quedi més professional.

Aquí és on s'ha vist que la decisió d'escollar Python com a llenguatge de programació ha estat la correcta, ja que aquesta connexió client-servidor, s'ha pogut de manera molt simple i no ha causat els mals de cap que un altre llenguatge podria haver creat.

7.5 Estructura del model

Pel que fa al model, com s'ha comentat la idea bé del Paper de St. Andrews i la seva proposta a futur d'implementar-lo usant SMT. Tot i que hi ha conceptes que s'han usat al projecte que també es van usar al Paper, aquest usava una estructura multinivell que causava la comprovació de solucions redundants, per això s'ha decidit unificar tots els nivells en un per ajudar al solver a propagar millor. Aquesta decisió ha comportat que bàsicament tot el model sigui completament diferent i a més degut a la incorporació de variables reals gràcies a la tecnologia SMT, hi ha mecàniques que no es van poder implementar al Paper esmentat.

A banda l'objectiu d'optimització i les entrades del model respecte al treball de St. Andrews també han canviat. S'ha decidit que la quantitat d'objectes que s'han d'entrar no sigui un input del model i que sigui el solver el que hagi de decidir, ja que en algunes instàncies és difícil determinar la quantitat que ha d'entrar de cada tipus així que s'ha decidit millor delegar la responsabilitat al solver.

A més com que s'ha pogut implementar de manera precisa la quantitat d'objectes a cada casella del *blueprint*, s'ha decidit implementar una optimització addicional que redueix la pèrdua d'objectes al *blueprint*, on pèrdua s'entén com la quantitat d'objectes que no s'estan aprofitant per produir altres objectes. Finalment, com a extra també s'ha afegit la possibilitat d'optimitzar la quantitat de cintes i *inserters* que s'usen per transportar objectes.

7.6 Instàncies

Les entrades del model permeten crear moltíssimes instàncies, però per avaluar el rendiment s'ha decidit seguir un patró força estàndard, el que s'ha fet ha sigut separar les instàncies per mida de *blueprint* des de 5x5 fins a 8x8. A més per cada mida s'han creat instàncies per diferents receptes les quals s'han ordenat de menys a més materials d'entrada requerits. Finalment, un cop feta la divisió entre mides i recepta, s'han creat múltiples instàncies variant les posicions per on entren els diferents materials i per on ha de sortir el material de la recepta objectiu.

Capítol 8

Disseny del model

En aquest apartat s'expliquen totes les restriccions que conformen el model, des de la primera iteració del model, juntament amb les millors de comportament del model i les millors de rendiment aplicades.

8.1 Implementació del model base

8.1.1 Rutes

Per implementar la noció de ruta s'ha usat la representació incremental on un element que forma part d'una ruta ha de precedir un element amb un valor de ruta inferior a ell i ser precedit per un element amb valor de ruta superior a ell. Per implementar aquesta representació s'ha creat una variable de tipus matriu `route` de mida $width \times height$, el seu domini també és $width \times height$, ja que una ruta pot ocupar com a màxim tota l'àrea del *blueprint*, a banda també cal saber l'orientació dels elements que formen part de la ruta (`inserters` i cintes) les quals es guarden en dues variables una per cada element `conveyor` i `inserter` aquestes de la mateixa manera que la ruta són matrius de mida $width \times height$ i el seu domini $[empty, north, east, south, west]$ on *empty* significa que no hi ha cap `inserter` o cinta present. Amb aquestes variables ja es poden definir les restriccions que conformen una ruta, principalment en són dues:

Augment de ruta

Aquesta restricció ens codifica qualsevol element que formi part d'una ruta, ha de tenir una casella adjacent la qual el seu valor de la ruta sigui superior. En aquest cas les caselles adjacents vàlides només són les que estan en la mateixa direcció que la cinta o `inserter` corresponent a la posició de la ruta. A banda també s'ha de tenir en compte que les rutes poden acabar si a la posició de la ruta hi ha un `inserter` i en la direcció on aquest apunta hi ha un `assembler`. A banda una ruta també pot acabar si la posició de la ruta es tracta d'una casella output donada com a entrada del model. En aquests dos casos no cal assegurar que la posició en la direcció del `inserter` el valor de ruta sigui més gran. Finalment, la implementació de la restricció és la següent.

LISTING 8.1: Forward Consistency

```

1 for each cell (i, j) in the grid of size (height x width):
2     if not is_output(i, j):
3         inserter_output = []
4         conveyor_output = []
5         for each dir in [north, east, south, west]:

```

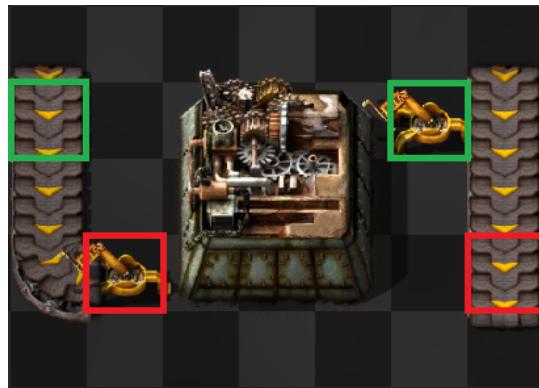


FIGURA 8.1: Possibles inicis "verd" i finals "vermell" de rutes

```

6      x = i + displacement[dir][0]
7      y = j + displacement[dir][1]
8      if 0 <= x < height and 0 <= y < width:
9          conveyor_output += If(conveyor[i][j] == dir,
10                         route[x][y] > route[i][j],
11                         False)
12
13      inserter_output += If(And(inserter[i][j] == dir,
14                                assembler[x][y] == 0),
15                                route[x][y] > route[i][j],
16                                False)
17
18      inserter_output += If(And(inserter[i][j] == dir,
19                                assembler[x][y] != 0),
20                                route[x][y] == 0,
21                                False)
22
23      assert Implies(route[i][j] > 0, Or(conveyor_output +
inserter_output))

```

Cal mencionar que si les rutes només haguessin d'anar d'un punt A a un punt B amb aquesta restricció seria suficient per assegurar que no es creïn cicles i que realment s'està creant una ruta que va del punt A al B, però com que en el nostre cas les rutes s'han de poder bifurcar, unir i arribar des de múltiples punts a múltiples punts, cal la següent restricció.

Decrement de ruta

Per poder incloure bifurcations i unions ens cal una restricció que de manera similar a l'anterior restricció ens asseguri que si una cinta o *inserter* forma part d'una ruta i no es tracta d'un inici de ruta, llavors ha de tenir en una de les caselles veïnes, hi hagi un element de la ruta amb un valor inferior. Aquestes caselles veïnes seran diferents en funció de si l'element de la ruta tracta d'una cinta o un *inserter*. Al cas de la cinta aquesta pot rebre input de qualsevol direcció que no sigui la mateixa que ella (Figura: ??) i en cas del *inserter* aquest només pot rebre input de la casella en la direcció contrària al *inserter* (Figura: ??), a banda els *inserters* també poden agafar objectes a un *assembler*, així que en aquest cas forçarem que el valor de ruta a la posició del *inserter* sigui 1 (inici de ruta). Així doncs, la restricció queda de la següent manera:

LISTING 8.2: Backwards Consistency

```

1 for each cell (i, j) in the grid of size (height x width):
2   if not is_input(i, j):
3     inserter_input = []
4     conveyor_input = []
5     for each dir in [north, east, south, west]:
6       x = i + displacement[dir][0]
7       y = j + displacement[dir][1]
8       if 0 <= x < height and 0 <= y < width:
9         conveyor_input += If(And(conveyor[i][j] != dir,
10                           conveyor[i][j] != empty),
11                           And(route[x][y] < route[i][j],
12                               route[x][y] > 0),
13                           False)
14
15         inserter_input += If(And(insert[i][j] == opposite(dir),
16                               assembler[x][y] == 0),
17                               And(route[x][y] < route[i][j],
18                                   route[x][y] > 0),
19                               False)
20
21         inserter_input += If(And(insert[i][j] == opposite(dir),
22                               assembler[x][y] != 0),
23                               route[i][j] == 1,
24                               False)
25
26 assert Implies(route[i][j] > 0, Or(conveyor_input + inserter_input))

```

Amb aquestes dues restriccions assegurem que la ruta no generi bucles i arribi als punts corresponents, però no ens assegura que l'orientació dels elements ni quins elements poden estar interconnectats entre si, així doncs per acabar de definir una ruta cal afegir quines entrades i sortides són vàlides per una cinta i un *inserter*:

Entrada i sortida de les cintes

Amb aquestes dues restriccions definim quines entrades i sortides són vàlides per una cinta. Pel que fa a les entrades pot rebre objectes per qualsevol de les posicions adjacents que no estiguin en la mateixa direcció que la mateixa cinta, a més aquestes caselles poden ser tant *inserters* com cintes. A més la cinta o *inserter* que estigui a la casella veïna ha d'apuntar a la cinta, és a dir, la seva direcció pot ser qualsevol menys l'oposada a la cinta. La restricció queda així:

LISTING 8.3: Conveyor Input

```

1 for each cell (i, j) in the grid of size (height x width):
2   if not is_input(i, j):
3     direction_clauses = []
4     for each dir in [north, east, south, west]:
5       x = i + displacement[dir][0]
6       y = j + displacement[dir][1]
7       if 0 <= x < height and 0 <= y < width:
8         direction_clauses += If(conveyor[i][j] != dir,
9                               Or(conveyor[x][y] == opposite(dir)
10
11                               ,
12                               inserter[x][y] == opposite(dir)
13                               False))
14
15 assert Implies(conveyor[i][j] != empty, Or(direction_clauses))

```

D'altra banda, una cinta només pot treure elements per la casella adjacent a la direcció a la qual apunta, aquesta casella veïna només pot estar ocupada per una cinta o un *insrter*, la direcció de la qual ha de ser qualsevol menys l'oposada a la cinta. La restricció queda de la següent manera:

LISTING 8.4: Conveyor Output

```

1 for each cell (i, j) in the grid of size (height x width):
2   if not is_output(i, j):
3     direction_clauses = []
4     for each dir in [north, east, south, west]:
5       x = i + displacement[dir][0]
6       y = j + displacement[dir][1]
7       if 0 <= x < height and 0 <= y < width:
8         direction_clauses += If(conveyor[i][j] == dir,
9                           Or(And(conveyor[x][y] != empty,
10                               conveyor[x][y] != opposite(
11                                 dir)),
12                               And(inserter[x][y] != empty,
13                                   inserter[x][y] == dir)),
14                               False))
15   assert Implies(conveyor[i][j] != empty, Or(direction_clauses))

```

Entrada i sortida dels inserters

Els *inserters* són molt similars a les cintes a l' hora de rebre objectes, però amb dues particularitats, primer només poden rebre objectes de la casella adjacent en la direcció contrària al *inserter* i segon, aquesta casella adjacent també pot ser un *assembler*. A l' hora de treure objectes de la mateixa manera que les cintes, només ho pot fer en la casella adjacent que es troba en la mateixa direcció, però en aquesta casella només hi pot haver una cinta que no apunti en la direcció oposada al *inserter* o bé un *assembler*. Les restriccions són les següents:

LISTING 8.5: Inserter Input

```

1 for each cell (i, j) in the grid of size (height x width):
2     if not is_input(i, j):
3         direction_clauses = []
4         for each dir in [north, east, south, west]:
5             x = i + displacement[dir][0]
6             y = j + displacement[dir][1]
7             if 0 <= x < height and 0 <= y < width:
8                 if not is_output(x, y):
9                     direction_clauses += If(insertter[i][j] == opposite(dir)
10
11
12
13 assert Implies(insertter[i][j] != empty, Or(direction_clauses))

```

LISTING 8.6: Inserter Output

```

10                     conveyor[x][y] != opposite(
11             dir)),
12             assembler[x][y] != 0),
13     False))
assert Implies(inserter[i][j] != empty, Or(direction_clauses))

```

Amb totes aquestes restriccions ja queden definides les rutes, tot seguit alguns exemples senzills i l'assignació de les variables `route` i `inserter` i `conveyor` de l'exemple 5.9:

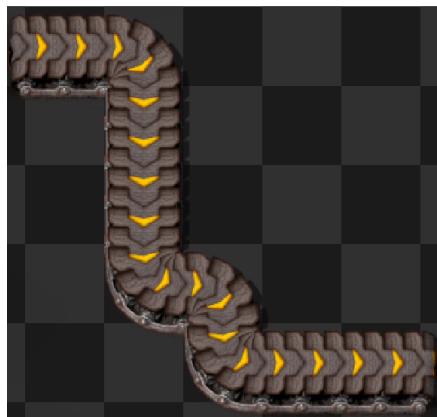


FIGURA 8.2: Representació gràfica de la variable `conveyor`

1	6	0	0	0
0	8	0	0	0
0	13	0	0	0
0	18	19	0	0
0	0	21	22	23

FIGURA 8.3: Model de la variable `route`

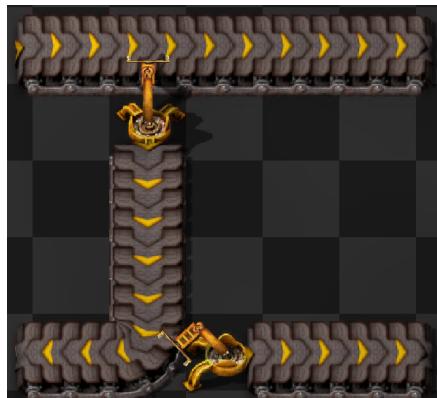


FIGURA 8.4: Representació gràfica de la variable `conveyor`

1	2	22	23	24
0	3	0	0	0
0	4	0	0	0
0	7	0	0	0
14	12	15	20	23

FIGURA 8.5: Model de la variable `route`

8.1.2 Restriccions dels *assemblers*

A l'hora de representar els *assemblers* hem de tenir en compte que ocupen un espai de 3x3 caselles, això complica la detecció de col·lisions entre ells i entre la resta


```

11             assembler[x][y] != 0),
12             False))
13     assert Implies(insert[i][j] != empty, Or(direction_clauses))

```

Així i tot, amb això no n'hi ha prou, ja que no estem forçant a les caselles on no hi ha *assemblers* que prenguin el valor 0, impedint així la utilització d'aquesta variable per saber on hi ha exactament un **assembler**.

0	0	0
0	0	0
0	0	0

FIGURA 8.7: Variable **assembler** sense cap *assembler* present

0	0	0	0	0
0	1	1	0	0
1	0	0	0	0
1	1	0	1	0
0	0	1	0	0

FIGURA 8.8: Variable **collision** prenen valors > 0 on no hi ha *assemblers*

Link assembler collision

Amb aquesta restricció hem d'aconseguir que **collision** només pregui valors > 0 on hi ha un assembler, per a fer-ho només cal assegurar que per cada posició de la matriu de la variable **collision** si hi ha una casella que ha pres valor > 0 llavors una de les 8 caselles adjacents a la variable **assembler** ha de tenir un valor > 0 .

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	1	0	0	0	4
0	0	0	0	0	0
0	0	0	0	0	0

FIGURA 8.9: Variable **assembler** amb 2 assignacions > 0

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	1	1	0	4	4	4
0	1	1	1	0	4	4	4
0	1	1	1	0	4	4	4
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

FIGURA 8.10: Variable **collision** prenen valors a l'àrea 3×3 només on hi ha presents *assemblers*

LISTING 8.8: Link Assembler Collision

```

1 for each cell (i, j) in the grid of size (height x width):
2     neighbors = []
3     for each neighbor (di, dj) in the 3x3 grid centered at (i, j):
4         x = di + i - 1
5         y = dj + j - 1
6         if 0 <= x < placement_height and 0 <= y < placement_width:
7             neighbors += assembler[x][y] == collision_area[i][j]
8     assert Implies(collision_area[i][j] != 0,
9                     Or(neighbors))

```

8.1.3 Tipus d'objectes

Per modelar quin tipus d'objectes passen per cada casella s'ha usat la variable **item_flow** que tracta d'una matriu de mida $width \times height$ amb domini $[0..max_items]$. **max_items** és un upper bound que es pot calcular a partir de les entrades del model, concretament donat l'ítem de sortida que el *blueprint* ha de produir, podem analitzar quines receptes el produeixen i quins elements requereixen aquestes receptes així recursivament fins que només quedin objectes bàsics que no s'obtenen com a sortida d'una recepta.

Les restriccions associades al flux d'objectes són les següents:

Part of route

Amb aquesta restricció fem que per totes les posicions **i j** de la variable **route** que prenguin un valor superior a 0, és a dir que formen part d'una ruta, forcen que a la mateixa posició **i j** de la variable **item_flow** el valor pres sigui superior a 0, és a dir que transporten algun objecte.

LISTING 8.9: Part of Route

```

1 for each cell (i, j) in the grid of size (height x width):
2     assert (route[i][j] > 0) == (item_flow[i][j] > 0)

```

Entrada i sortida d'objectes

Com s'ha comentat a l'apartat ?? una de les entrades del model tracta d'especificar quins objectes hi ha d'haver a les caselles d'entrada i sortida. Aquesta restricció, doncs, s'encarrega d'assegurar que aquestes caselles portin els objectes especificats.

LISTING 8.10: Item Input i Output

```

1 for each cell (i, j) in input_cells:
2     assert item_flow[i][j] == input_item(i, j)
3
4 for each cell (i, j) in output_cells:
5     assert item_flow[i][j] == output_item(i, j)

```

Propagació del tipus d'objectes

Finalment, la restricció més important és la que s'encarrega d'assegurar que els objectes específicats a les entrades, sortides i els objectes requerits i produïts per les receptes associades al *assemblers*, es distribueixen correctament per les rutes.

Per aconseguir aquest comportament els que s'ha fet és, per totes les posicions del *blueprint*, i cada posició ortogonalment veïna a la casella que estigui dins el *blueprint*, si la casella té un valor a la variable **route** > 0 , la direcció on es troba respecte a la direcció de la casella central és l'oposada i la casella central tracta d'un *inserter*, llavors el valor de la variable **item_flow** a la posició del *inserter* ha de ser la mateixa que la de la posició veïna.

D'altra banda, i de manera molt similar si la casella ortogonalment veïna té un valor a la variable **route** > 0 i es troba en la mateixa direcció que l'*inserter* central llavors és la casella veïna que ha de tenir el mateix valor a **item_flow** que la casella central. Finalment, si la casella central tracta d'una cinta, només cal assegurar que la casella veïna que es troba en la mateixa direcció que la cinta central ha de tenir el mateix valor a **item_flow** que la cinta.

LISTING 8.11: Item Carry

```

1 for each cell (i, j) in the grid of size (height x width):
2     inserter_carry = []
3     conveyor_carry = []
4     for each dir in [north, east, south, west]:
5         x = i + displacement[dir][0]
6         y = j + displacement[dir][1]
7         if 0 <= x < height and 0 <= y < width:
8             if not is_input(i, j):
9                 inserter_carry += Implies(And(inserter[i][j] == opposite(
10                dir),
11                route[x][y] > 0),
12                item_flow[i][j] == item_flow[x][
13                y])
14                inserter_carry += Implies(And(inserter[i][j] == dir,
15                route[x][y] > 0),
16                item_flow[x][y] == item_flow[i][
17                j])
18                if not is_output(i, j):
19                    conveyor_carry += Implies(conveyor[i][j] == dir,
20                        item_flow[x][y] == item_flow[i][
21                        j])
22    assert Implies(inserter[i][j] != empty, And(inserter_carry))

```

19 assert Implies(conveyor[i][j] != empty, And(conveyor_carry))

Alguns detalls importants sobre la restricció són:

Només cal tenir en compte que la casella veïna tingui un valor a la variable **item_flow** > 0 si la casella central tracta d'un *inserter*, ja que són els únics elements d'una ruta que a la seva entrada o sortida hi pot haver un *assembler* és a dir una posició on mai hi pot haver un valor d'**item_flow** > 0.

Les caselles de sortida del *blueprint*, que només poden ser cintes, no han de propagar el seu objecte, d'aquí la comprovació de **if not is_output(i, j)**.

Tot i que no hi pot haver *inserters* a les posicions d'entrada d'objectes, s'ha afegit la comprovació de **if not is_input(i, j)**, ja que ens estalvia afegir restriccions addicionals.

Les implicacions finals són redundants, ja que només iterem per les direccions [**north, east, south, west**] i les implicacions anteriors asseguren que les direccions de les cintes i *inserters* siguin la mateixa que iterem o l'oposada, les quals mai seran **empty**, així i tot, les implicacions finals ajuden a reduir el temps de solving, d'alguna manera deuen estar ajudant al solver a propagar més fàcilment les implicacions anteriors.

8.1.4 Quantitat d'objectes

Per determinar la quantitat d'objectes que passen per una casella, he decidit usar la mesura d'objectes per minut, el motiu és perquè té un bon balanç entre precisió i eficiència a l'hora de definir el domini de la variable.

Modelar aquesta mecànica del joc és de les més costoses i difícils de replicar, tot seguit explico les variables usades, el raonament els pros i contres.

Per cada casella cal saber quants objectes entren i quants surten, per poder modelar aquests objectes se sumen i reparteixen. Així doncs, s'han usat dues variables: **input_flow_rate** i **output_flow_rate** que són dues matrius de mida $width \times height$ i els seus dominis [0..450] on 450 és el màxim nombre d'objectes per minut que un element de la ruta pot dur, en aquest cas la cinta, així que s'haurà d'anar amb compte i assegurar que els *inserters* duguin més objectes per minut que la seva capacitat màxima de 50.

Les variables **output_flow_rate** i **input_flow_rate** són de tipus real, ja que les ràtios d'entrada/sortida de moltes receptes fan que una entrada entera d'objectes minut produueixi un nombre decimal d'objectes de sortida, i arrodonir aquests representa perdre molta precisió i allunyar-se dels casos reals que es podrien assolir amb les mecàniques originals del joc.

Les restriccions associades a aquestes variables que modelen la quantitat d'objectes que circulen pel *blueprint* són les següents:

Part of route

De manera molt similar a la restricció Part of route, qualsevol posició **i j** del *blueprint* que no formi part de la ruta no pot dur cap mena d'objecte i com a conseqüència

el nombre d'objectes per minut que transporta ha de ser 0.

LISTING 8.12: Part of Route

```

1 for each cell (i, j) in the grid of size (height x width):
2     assert Implies(route[i][j] == 0,
3                     And(input_flow_rate[i][j] == 0,
4                         output_flow_rate[i][j] == 0))

```

Quantitat d'entrada

Una de les dades que tenim del *blueprint* és la quantitat d'objectes per minut que hi ha a les caselles d'entrada, així que aquesta restricció assegura que el valor de la variable **input_flow_rate** a les posicions **i** **j** d'entrada, és l'especificada pel *blueprint*.

LISTING 8.13: Item Input Rate

```

1 for each cell (i, j) in input_cells:
2     assert input_flow_rate[i][j] == input_rate[i][j]

```

Propagació de la quantitat d'objectes (Cintes)

La part més important de modelar la quantitat d'objectes que circulen per una cel·la que forma part de la ruta és definir quina serà l'entrada i sortida d'objectes en funció dels elements adjacents. Com el model consta de cintes i *inserters* els quals tenen comportaments bastant diferents, la propagació de la quantitat d'objectes s'ha separat per cintes i *inserters*. Pel que fa a les cintes hi ha dues normes que defineixen com es distribueix la quantitat d'objectes:

Per cada posició **i** **j** del *blueprint* on hi hagi una cinta, el valor de la variable **input_flow_rate** a la posició **i** **j** ha de ser la suma dels valors de la variable **output_flow_rate** a les posicions ortogonalment adjacents, les quals hi hagi un element de ruta, tant un *insrter* com una altra cinta, on la seva direcció apunti a qualsevol de les 3 entrades de la cinta central. És a dir l'entrada d'una cinta és la suma de sortides dels elements de la ruta que aporten objectes a aquesta cinta.

Paral·lelament, per cada posició **i** **j** del *blueprint* on hi hagi una cinta, el valor de la variable **output_flow_rate** a la posició **i** **j** serà el valor la variable **input_flow_rate** de la mateixa cinta, menys la suma de **input_flow_rate** d'*inserters* a les posicions adjacents a la cinta, els quals treguin elements de la dita cinta, és a dir que la seva direcció sigui la mateixa a la direcció de la casella adjacent a la cinta central.

Tot seguit un exemple visual per entendre bé el que s'ha descrit.

LISTING 8.14: Belt Item Flow Propagation

```

1 belt_flow_rate_propagation = []
2 for each cell (i, j) in the grid of size (height x width):
3     belt_input = []
4     belt_output = []
5     for each dir in [north, east, south, west]:
6         x = i + displacement[dir][0]
7         y = j + displacement[dir][1]
8         if 0 <= x < height and 0 <= y < width:
9             belt_input += (If(And(conveyor[i][j] != dir,
10                                Or(conveyor[x][y] == opposite(
11                                dir),
12                                inserter[x][y] == opposite(
13                                dir))),,

```

```

12                     output_flow_rate[x][y], 0))
13
14             belt_output += (If(inserter[x][y] == dir,
15                             input_flow_rate[x][y], 0))
16
17     if not is_input(i, j):
18         assert Implies(conveyor[i][j] != empty,
19                         And(input_flow_rate[i][j] == sum(belt_input),
20                             input_flow_rate[i][j] <= 450))
21
22     assert Implies(conveyor[i][j] != empty,
23                     output_flow_rate[i][j] ==
24                     (input_flow_rate[i][j] - sum(belt_output)))

```

Propagació de la quantitat d'objectes (Inserters)

Pel que fa als *inserters*, la propagació de la quantitat d'objectes és lleugerament diferent de la de les cintes, ja que la seva capacitat de transport és de tan sols 50 objectes per minut. A banda la quantitat d'objectes que un *inserter* agafa depèn de si l'*inserter* està agafant objectes d'una cinta o si es tracta d'un *inserter* de sortida d'un *assembler*, així doncs es requereixen tres restriccions per modelar el comportament dels *inserters*.

Per cada posició i j del *blueprint* on hi hagi un inserter, és a dir on la variable **inserter[i][j]** ≠ **empty**, i per cada casella x y ortogonalment adjacents a l'*inserter*, si la casella veïna està en la direcció oposada al *inserter* i aquesta té un valor a la variable **input_flow_rate[x][y]** ≥ 50, llavors el valor d'entrada **input_flow_rate** i de sortida **output_flow_rate** l'*inserter* haurà de ser 50, ja que un *inserter* sempre que pugui agafarà el màxim d'objectes disponibles de la casella de la qual se supleix.

En cas que el valor de **input_flow_rate[x][y]** < 50 i el valor de ruta a la posició **route[x][y]** ≠ 0, significa que l'*inserter* està prenent objectes d'una cinta la qual la seva entrada és inferior a 50 objectes minut, per tant, el valor d'entrada **input_flow_rate** i de sortida **output_flow_rate** l'*inserter* haurà de ser igual que el valor d'entrada de la casella veïna.

Finalment, en cas que el valor de ruta a la casella veïna sigui **route[x][y]** = 0 significa que l'*inserter* està traient objectes produïts per la recepta d'un *assembler*, en aquest cas no es pot especificar la quantitat d'entrada o sortida de l'*inserter*, així que només assegurem que el possible valor que la recepta del *assembler* assigni a l'*inserter* estigui dins el rang de transport, $0 \geq \text{input_flow_rate}[i][j] \geq 50$ i $0 \geq \text{output_flow_rate}[i][j] \geq 50$.

LISTING 8.15: Inserter Item Flow Propagation

```

1 for each cell (i, j) in the grid of size (height x width):
2     inserter_input = []
3     for each dir in [north, east, south, west]:
4         x = i + displacement[dir][0]
5         y = j + displacement[dir][1]
6         if 0 <= x < height and 0 <= y < width:
7             inserter_input += Implies(And(insert[i][j] == opposite(
8                 dir),
9

```

```

9                                         And(input_flow_rate[i][j] == 50,
10                                         output_flow_rate[i][j] ==
11                                         50))
12                                         inserter_input += Implies(And(inserter[i][j] == opposite(
13                                         dir),
14                                         input_flow_rate[x][y] < 50,
15                                         route[x][y] != 0),
16                                         And(input_flow_rate[i][j] ==
17                                         input_flow_rate[x][y],
18                                         output_flow_rate[i][j] ==
19                                         input_flow_rate[x][y]))
20                                         inserter_input += Implies(And(inserter[i][j] == opposite(
21                                         dir),
22                                         route[x][y] == 0),
23                                         And(input_flow_rate[i][j] ==
24                                         output_flow_rate[i][j],
25                                         input_flow_rate[i][j]<=50,
26                                         input_flow_rate[i][j]>=0,
27                                         output_flow_rate[i][j]<=50,
28                                         self.output_flow_rate[i][j]
29                                         ]>=0))
30
31     assert Implies(inserter[i][j] != empty, And(inserter_input))

```

8.1.5 Receptes

Juntament amb la quantitat d'objectes, les receptes són una de les parts més importants del model. Les receptes requereixen totes les anteriors restriccions descrites, ja que necessiten cert tipus d'objectes com a entrada, en funció de la quantitat en produueixen una quantitat de sortida, es produueixen en els *assemblers* i l'única manera de fer-los arribar és mitjançant rutes.

Per modelar les receptes, primer necessitem associar una recepta a un *assembler*, això s'ha fet usant la variable **selected_recipe**, aquesta variable tracta d'un array tipus BitVector, la seva mida és de **max_assemblers** que com s'ha explicat anteriorment es calcula fent $(width/3) \times (height/3)$. El domini de la variable és $[0..max_recipes]$, on **max_recipes** tracta d'un upper bound que es calcula de manera molt similar a **max_items**, donat l'objecte que es vol fabricar al *blueprint* es pot saber quina recepta el fabrica i els objectes que necessita, recursivament podem saber quantes receptes pengen de la recepta que produueix la recepta final i establir l'upper bound.

Un cop podem saber quina recepta té associada cada *assembler*, ens cal assegurar que a l'*assembler* només entren i surten els objectes requerits per la recepta que té associada. Per definir aquest comportament no ha sigut necessària cap variable més així que tot seguit s'explicarà les restriccions associades i les variables implicades anteriorment descrites.

Finalment, hem d'assegurar que l'*assembler* està produint la quantitat correcta d'objectes en funció de la recepta seleccionada i la quantitat d'objectes entrants. Aquesta és la part més complexa de les receptes, ja que els objectes poden entrar a l'*assembler*

en diferents ràtios i com ja s'ha explicat a l'apartat [Receptes](#), hi ha una quantitat màxima d'objectes entrants els quals la recepta pot processar.

La idea per modelar aquests comportaments ha sigut la següent:

Per decidir quina ha de ser la quantitat d'objectes de sortida, per cada *assembler* ens cal saber les ràtios d'entrada de la recepta, aquesta informació la tindrem a la variable `input_ratio` que tracta d'una matriu de mida `max_assemblers × max_items`, de domini [0..1] i de tipus Real.

Amb les ràtios de cada objecte d'entrada per cada *assembler*, necessitem saber quina és la mínima ràtio per així saber quina és la quantitat d'objectes màxima que pot generar l'*assembler* amb els ingredients d'entrada. Així doncs, la ràtio mínima es guarda en una variable auxiliar `min_ratio` que tracta d'un array de mida `max_assemblers`, de tipus Real i el seu domini igual que `input_ratio` és [0..1].

Amb les noves variables que formen part de les receptes, les restriccions que les utilitzen són les següents:

Associar recepta

Amb aquesta restricció s'assegura que cada *assembler* que formi part del *blueprint* tingui una recepta associada, això es fa de la següent manera. Per cada *assembler* `k [1..max_assemblers]` i cada posició `i j` on hi pot haver un *assembler* (`width-2 × height-2`), es mira si la variable `assembler[i][j]=k`, i si per alguna de les posicions `i j` es dona tal condició llavors la variable `selected_recipe[k]` ha de ser diferent de 0, és a dir té una recepta vàlida associada, d'altra banda, si no es troba cap posició `i j` on `assembler[i][j]=k` llavors la variable `selected_recipe[k]` ha de ser 0, és a dir que no té cap recepta associada.

LISTING 8.16: Associate Recipe

```

1 for k in [1..max_assemblers]
2     exists_assembler = []
3     for each cell (i, j) in the grid of size (height-2 x width-2):
4         exists_assembler += assembler[i][j] == k
5     assert If(Or(exists_assembler),
6             selected_recipe[k - 1] != 0,
7             selected_recipe[k - 1] == 0)

```

Ingredients d'entrada i sortida

Per la recepta associada a un *assembler* hem d'assegurar que els objectes que entren només siguin els que formen part dels ingredients d'entrada de la recepta. Per assegurar aquest comportament cal: Per cada *assembler* del *blueprint* i cada *inserter* amb direcció apuntant al *assembler*, hem d'assegurar que com a mínim hi ha un *inserter* per cada tipus d'objecte que la recepta requereix i que no hi ha cap *inserter* que dugui un objecte no requerit per la recepta.

De la mateixa manera hem d'assegurar que hi hagi com a mínim un *inserter* que s'endugui l'objecte que la recepta produeix.

LISTING 8.17: Assembler Input

```

1 for each cell (i, j) in the grid of size (height-2 x width-2):
2     for k in [1..max_assemblers]:
3         assembler_selected = assembler[i][j] == k
4         for item in [1..max_items]:

```

```

5     inputs = []
6     for dir in displacement:
7         for pos in displacement[dir]:
8             x = i + 1 + pos[0]
9             y = j + 1 + pos[1]
10            if 0 <= x < height and 0 <= y < width:
11                inputs += And(inserter[x][y] == opposite(dir),
12                               item_flow[x][y] == item)
13            for recipe in [1..max_recipes]:
14                recipe_selected = selected_recipe[k] == recipe
15                if recipe_input[recipe][item] != 0:
16                    assert Implies(And(assembler_selected, recipe_selected
17 ),
18                               Or(inputs))
19            else:
20                assert Implies(And(assembler_selected, recipe_selected
21 ),
22                               Not(Or(inputs)))

```

LISTING 8.18: Assembler Output

```

1 for each cell (i, j) in the grid of size (height-2 x width-2):
2     for k in [1..max_assemblers]:
3         assembler_selected = assembler[i][j] == k
4         for item in [1..max_items]:
5             outputs = []
6             for dir in displacement:
7                 for pos in displacement[dir]:
8                     x = i + 1 + pos[0]
9                     y = j + 1 + pos[1]
10                    if 0 <= x < height and 0 <= y < width:
11                        outputs += And(inserter[x][y] == dir,
12                                       item_flow[x][y] == item)
13                    for recipe in [1..max_recipes]:
14                        recipe_selected = selected_recipe[k] == recipe
15                        if recipe_output[recipe][item] != 0:
16                            assert Implies(And(assembler_selected, recipe_selected
17 ),
18                               Or(outputs))
19            else:
20                assert Implies(And(assembler_selected, recipe_selected
21 ),
22                               Not(Or(outputs)))

```

Alguns detalls importants de les restriccions són, primer **displacement** tracta d'un diccionari Python amb quatre entrades representant cada direcció cardinal numerada de [1..4] i per cada direcció hi ha una llista amb les parelles de coordenades **x** **y** representant totes les caselles relatives a (0,0) que es consideren vàlides com a casella d'entrada o sortida d'un *assembler*.

Segon, tot i que Z3 permet usar Arrays indexables per valors de variables enteres, aquests introduixen noves teories al model que alenteixen el temps de solving, per això aquesta restricció ha d'iterar per cada *assembler*, recepta i objecte. I usar dues variables auxiliars (**assembler_selected**, **recipe_selected**) per saber si l'*assembler* que estem iterant tracta del *assembler* a la posició **i** **j** i si la recepta que té associada tracta de la recepta que estem iterant.

Per últim, **recipe_input** i **recipe_output** són dues matrius Python que s'obtenen del preprocès de les entrades del *blueprint*, concretament cada fila representa una recepta, cada columna un objecte i cada posició conté quants objectes per minut

usa aquella recepta en cas de **recipe_input** i quants objectes produeix en cas de **recipe_output**.

Ràtios d'entrada

Com s'ha comentat anteriorment per determinar quants objectes ha de produir una recepta necessitem saber les ràtios dels objectes d'entrada, aquesta restricció s'encarrega justament d'això.

La implementació és molt similar a les restriccions d'ingredients d'entrada i sortida, per cada *assembler* a la posició **i j** del *blueprint* i cada posició veïna vàlida per l'entrada d'objectes mitjançant *inserters*, ens guardem la suma de **output_flow_rate[i][j]** dels *inserters* que portin el mateix tipus d'objecte, gràcies al fet que la restricció anterior ens assegura que només hi hagi *inserters* afegint objectes que formen part de la recepta, no cal comprovar el tipus d'objecte que transporta l'*inserter*, només la quantitat. Aquesta suma l'hem de dividir pel màxim nombre d'objectes que la recepta accepta d'aquell tipus (**recipe_input[recepta][objecte]**) obtenint així la ràtio d'entrada de cada objecte, aquesta ràtio la guardem a la variable **input_ratio[assembler][objecte]**

LISTING 8.19: Input Ratio

```

1 for each cell (i, j) in the grid of size (height-2 x width-2):
2     for k in [1..max_assemblers]:
3         assembler_selected = assembler[i][j] == k
4         for item in [1..max_items]:
5             inputs = []
6             for dir in displacement:
7                 for pos in displacement[dir]:
8                     x = i + 1 + pos[0]
9                     y = j + 1 + pos[1]
10                    if 0 <= x < height and 0 <= y < width:
11                        inputs += If(And(insert[x][y] == opposite(dir),
12                                         item_flow[x][y] == item),
13                                         output_flow_rate[x][y], 0)
14         for recipe in [1..max_recipes]:
15             recipe_selected = selected_recipe[k] == recipe
16             if recipe_input[recipe][item] != 0:
17                 assert Implies(And(assembler_selected,
18                                     recipe_selected),
19                                     input_ratio[k][item] ==
20                                     sum(inputs)/recipe_input[recipe][item])
21             else:
22                 assert Implies(And(assembler_selected,
23                                     recipe_selected),
24                                     input_ratio[k][item] == 1)

```

Un detall important de la implementació és que quan un objecte no s'usa com a ingredient en una recepta, el valor de la seva ràtio es posa a 1, això és important, ja que per la següent restricció ens serà molt útil.

Ràtio mínima

Amb les ràtios de cada objecte per cada *assembler*, només ens queda saber quin és el mínim, això s'aconsegueix usant la variable auxiliar **min_ratio** anteriorment descrita. Per forcar que el valor que prengui la variable **min_ratio** sigui el mínim de cada objecte del *assembler*, és molt simple, per cada *assembler* **k** i cada objecte **i**, cal que el valor de **min_ratio[k]** sigui un valor de qualsevol objecte de la variable que aparegui **input_ratio[k]** i a més també cal forçar que el valor de **min_ratio[k]**

sigui més petit igual a tots els valors dels objectes de `input_ratio[k]`. D'aquesta manera al valor haurà de ser el més petit i formar part dels possibles valors de les ràtios.

LISTING 8.20: Min Ratio

```

1 for k in [1..max_assemblers]:
2     value_of = []
3     min_value = []
4     for item in [1..max_items]:
5         value_of += min_ratio[k] == input_ratio[k][item]
6         min_value += min_ratio[k] <= input_ratio[k][item]
7     assert Or(value_of)
8     assert And(min_value)

```

Quantitat d'objectes de sortida

Amb totes les peces del puzzle llestes, només cal definir quants objectes per minut ha de produir la recepta associada al *assembler*.

De manera similar a anteriors restriccions, per cada *assembler* del *blueprint* i per cada posició vàlida d'entrada i sortida d'objectes del *assembler* sumem el nombre d'objectes per minut que els *inserters* estan extraient. Aquesta suma ha de ser igual al valor de la variable `min_ratio` a la columna del *assembler* que estiguem mirant multiplicat pel màxim teòric que pot produir la recepta `recipe_output[recepta][objecte]`.

LISTING 8.21: Min Ratio

```

1 for each cell (i, j) in the grid of size (height-2 x width-2):
2     for k in [1..max_assemblers]:
3         assembler_selected = assembler[i][j] == k + 1
4         outputs = []
5         for dir in displacement:
6             for pos in displacement[dir]:
7                 x = i + 1 + pos[0]
8                 y = j + 1 + pos[1]
9                 if 0 <= x < height and 0 <= y < width:
10                     outputs += If(inserter[x][y] == dir,
11                               output_flow_rate[x][y],
12                               0))
13         for recipe in [1..max_recipes]:
14             recipe_selected = selected_recipe[k] == recipe + 1
15             for item in [1..max_items]:
16                 if recipe_output[recipe][item] != 0:
17                     assert Implies(And(assembler_selected,
18                                         recipe_selected),
19                                         sum(outputs) == min_ratio[assembler] *
20                                         recipe_output[recipe][item]))

```

8.2 Canvis fets al model per aproximar-lo més al comportament real del joc

La primera iteració del model és completament funcional, però hi ha alguns comportaments que no acaben de ser com al Factorio. Tot seguit s'explica quins d'aquests comportaments es poden extrapolar del joc al model i quins requeririen massa complexitat i s'han deixat de banda.

8.2.1 Propagació de la quantitat d'objectes en inserters

Com s'ha explicat a l'apartat de la implementació d'aquesta mecànica, els inserters sempre intenten transportar el màxim d'objectes possible, però amb la representació d'aquest comportament a l'anterior restricció, els inserters només podrien prendre dos valors:

En cas que la sortida de la cinta de la qual agafen objectes fos superior a 50 aquests agafaven els 50 objectes/min, i en cas que l'output de la cinta fos inferior a 50 objectes per minut, l'inserter agafa la quantitat de sortida de la cinta, definint d'aquesta manera el comportament de "sempre agafar el màxim possible". Però això no sempre és cert per dos motius:

Primer hi ha el cas que si l'inserter deixa els objectes en una altra cinta i aquesta porta una quantitat d'objectes que en afegir-se els que l'inserter porta se superi la capacitat de 450 objectes per segon, l'inserter hauria de deixar la diferència d'objectes fent així que el seu output no fos el màxim que pot agafar de la cinta d'entrada.

Segon si un inserter porta objectes a un assemblador i aquest no els necessita tots degut a que per la falta de quantitat d'un altre objecte no els pot processar, l'inserter acabarà reduint la quantitat d'objectes que pot aportar a l'assemblador fins que porti exactament la mateixa quantitat d'objectes que l'objecte de la recepta de l'assemblador que fa de coll d'ampolla, fent així que la seva quantitat de sortida no sigui la mateixa que la màxima que pot agafar de la cinta d'entrada.

Per poder recrear el comportament real s'ha optat per fer el següent:

En comptes de fixar el seu input a 50 objectes/min si sortida de la cinta és més gran o igual a 50, ara si la sortida de la cinta és més gran o igual a 50 llavors l'entrada de l'inserter haurà de ser com a molt 50 i com a molt poc més gran que 0, ja que si un inserter no transporta objectes és inútil.

D'altra banda, l'output de la cinta d'entrada de l'inserter és més petit que 50 objectes per minut, llavors l'input de l'inserter podrà ser com a màxim la quantitat de sortida de la cinta.

És important tenir en compte que aquesta nova manera de representar els objectes que poden transportar els inserters només representa en comportament real del joc, sempre que el criteri d'omplimentació sigui maximitzar la quantitat d'objectes que es volen produir, ja que si simplement volguéssim trobar una assignació que satisfés a totes les restriccions, ens podríem trobar casos on un inserter dins el llindar que s'ha establert a la nova restricció, no fos el màxim. Però com l'objectiu del model és maximitzar la sortida i aquest no té gaire sentit sense aquest criteri d'omplimentació, aquesta nova implementació s'ha considerat completament vàlida i funcional.

Arran d'aquest canvi, hi ha una altra part de les restriccions que també requereix un canvi, tot seguit s'explica a la següent secció.

8.2.2 Ràtios d'entrada i ràtio mínima

A la primera iteració del model es va explicar que per poder calcular la quantitat d'objectes que un assemblador ha de produir, calia prendre com a referència l'objecte que en relació amb la quantitat que necessita la recepta està sent el menys suplit pels inserters d'entrada. Però hi ha una part d'aquesta implementació que no s'ajusta del

tot a com funciona el joc, principalment es deu al fet que si hi ha un objecte que en funció de la seva necessitat a la recepta s'està suplint amb major quantitat que un altre, aquest acaba saturant al assembler fent que aquest al final només pugui processar la proporció relativa marcada per l'objecte que menys s'està suplint, és a dir si el model permet suprir més objectes dels que l'assembler pot processar aquests s'haurien d'acumular a la cinta que supleix a l'inserter de l'objecte esmentat, però amb l'antiga implementació aquest excés d'objectes simplement es volatilitzava pel fet que el que es feia era agafar la ràtio mínim d'entrada del assembler per calcular la quantitat de sortida.

Arreglar aquest comportament no ha estat gaire difícil i a més en certa manera ha ajudat al model a causa de l'eliminació de la variable `min_ratio`. El que s'ha fet és que com l'objecte que menys se supleix dictamina la quantitat dels altres objectes que l'assembler podrà processar, simplement s'ha eliminat la variable `min_ratio` i s'ha forçat que per cada objecte k a la variable `input_ratio[k]` els seus valors siguin iguals, d'aquesta manera cada inserter només podrà suprir a l'assembler la quantitat d'objectes que fa que la seva ràtio d'entrada sigui igual que la resta, fent així que no hi hagi objectes que es perdin.

La nova restricció que iguala els valors de les ràtios per cada assembler és la següent:

LISTING 8.22: Equal Ratios

```

1 def equal_ratios(self):
2     equal_ratios = []
3     for j in range(self.max_assemblers):
4         for i in range(1, self.max_items):
5             equal_ratios.append(
6                 self.input_ratio[j][i] == self.input_ratio[j][0]
7             )
8     return equal_ratios

```

8.2.3 Quantitat d'objectes de sortida

Finalment, l'últim comportament del joc que la primera iteració del model no replica, tracta de la quantitat d'objectes que han de treure els inserters d'un assembler. El primer model simplement assegura que la suma d'objectes que treuen els inserters de sortida siguin exactament la quantitat que l'assembler produeix, però no diu res de quants objectes ha de treure cada inserter. Això és un problema, ja que al joc si hi ha més d'un inserter que treu objectes d'un assembler la quantitat que aquest transporta ha de ser igual a la resta, així doncs el que s'ha fet és afegir una variable auxiliar a la restricció anterior que serveix perquè tots els inserters que treuen objectes de l'assembler tinguin el mateix valor, d'aquesta manera igualant la variable `output_flow_rate` a la posició x, y on es troba l'inserter a la variable s'assegura que tots els inserters que treuen objectes d'aquell assembler transportin la mateixa quantitat d'objectes mantenint la propietat d'assegurar que la suma d'objectes dels inserters sigui exactament igual a la produïda per l'assembler.

L'antiga restricció amb el canvi aplicat queda de la següent manera:

LISTING 8.23: Output Rate

```

1 def set_output_rate():
2     output_ratios = []

```

```

3   for i in range(placement_height):
4       for j in range(placement_width):
5           for assembler in range(max_assemblers):
6               outputs = []
7               output_rate = Real(f'output_rate_{i}_{j}_{assembler}')
8               output_ratios.append(And(output_rate >= 0, output_rate <=
50))
9               for direction in displacement:
10                  for pos in displacement[direction]:
11                      x = i + 1 + pos[0]
12                      y = j + 1 + pos[1]
13                      if 0 <= x < height and 0 <= y < width:
14                          outputs.append(
15                             If(inserter[x][y] == direction,
16                                output_flow_rate[x][y],
17                                0))
18               output_ratios.append(Implies(self.insertter[x][
19 y] == direction,
20                               output_flow_rate[
21 x][y] == output_rate))
22               first_item_ratio = next(iter(input_ratio[assembler + 1].
23 values()))
24               for output in recipes[selected_recipe[assembler]]["OUT"]:
25                   output_ratios.append(Implies(assembler[i][j] ==
assembler + 1,
26                                     sum(outputs) ==
first_item_ratio * output[1]))
27   return output_ratios
28

```

8.3 Millores al model

8.3.1 Upper bound

Amb les mecàniques del joc replicades el millor possible al model, hi ha certes millores que es poden fer. La més simple tracta d'acotar el domini $[0..width \times height]$ de la variable `route`, ja que si al blueprint no només hi ha cintes i inserters sinó que també necessitem assmblers els quals ocupen 3x3 caselles, així que el domini de `route` es pot reduir sabent el nombre de receptes `max_recipes` que implica crear un objecte determinat, com s'ha explicat anteriorment aquest valor es precalcula i és fàcil d'obtenir, finalment el nou domini de la variable serà $[0..width \times height - max_recipes \times 9]$.

8.3.2 Lower bound

Usant la mateixa idea d'abans, com se sap que la quantitat mínima d'assemblers que es necessitaran és igual al nombre de receptes diferents `max_recipes`, podem crear una restricció que aprofita aquesta informació i obligui que el nombre mínim d'assemblers presents al blueprint sigui `max_recipes`.

La implementació és molt senzilla i només cal que per cada posició `i j` de la variable `assembler` i assegurar que la suma de posicions on el valor d'assembler és superior a 0, és a dir hi ha un assembler, sigui com a mínim `max_recipes`.

LISTING 8.24: Lower Bound

```

1 def lower_bound_assemblers(self):
2     lower_bound = []
3     for i in range(self.placement_height):

```

```

4     for j in range(self.placement_width):
5         lower_bound.append(If(UGT(self.assembler[i][j], 0), 1, 0))
6     return [sum(lower_bound) >= self.max_recipes]

```

8.3.3 Computar les receptes associades als assemblers

Un dels inconvenients del model actual és que per saber quina recepta té associada cada assembler ens cal una variable nova la qual incorpora moltes simetries, ja que dos assemblers diferents amb receptes diferents es poden canviar de posició i canviar de recepta i obtenir el mateix resultat, és a dir si l'assembler 1 usa la recepta 2 a la posició (1, 1), i l'assembler 2 usa la recepta 1 a la posició (2, 2), és el mateix que si l'assembler 2 usa la recepta 2 a la posició (1, 1) i l'assembler 1 usa la recepta 1 a la posició (2, 2). El mateix passa si dos o més assemblers usen la mateixa recepta, aquests es poden permutar entre ells sense alterar el resultat.

La solució que s'ha proposat a aquest problema és bastant més complexa que les anteriors, ja que d'alguna manera cal saber quines receptes ha d'usar cada assembler abans d'iniciar el procés de solving, aquest problema no és gens trivial i s'ha optat per generar un sistema d'equacions el qual s'ha de resoldre maximitzant la variable que representa l'objecte de sortida. Tot seguit el desenvolupament d'un exemple per entendre correctament el sistema d'equacions.

Objectes i receptes

Suposem que l'objecte que es vol produir tracta de "advanced circuits", aquest objecte requereix la recepta Advanced-Circuit per ser produït. A partir d'aquesta recepta necessitem saber quins objectes necessita per ser fabricat que no siguin matèries primeres és a dir objectes que requereixen altres receptes per ser produïts. Concretament, necessita 40 copper cable/min i 20 electronic circuit/min així que aquests objectes impliquen les receptes de copper cable el qual es pot produir integradament a partir de matèries primeres i electronic circuit que requereix 360 copper/cable minut.

Així doncs, el desglossament de dependències de l'objecte advanced circuit en objectes és el següent:

Recepta	Requereix	Produeix
Advanced circuit	40 Copper Cable 20 Electronic Circuit	
Electronic Circuit	360 Copper Cable	120 Electronic Circuit
Copper Cable		240 Copper Cable

Sistema d'equacions

Amb aquesta informació podem construir un sistema d'equacions creant una variable real per cada recepta, aquesta variable representarà el percentatge d'*assemblers* que l'estan produint, és a dir 1 significa que hi ha un *assembler* funcionant al 100% i si és 1.2 significa que hi ha un *assembler* funcionant al 100% i un altre al 20%, ja que un *assembler* com a molt pot treballar al 100% de la seva capacitat.

Per aquest exemple tindrem tres variables `cca`, `eca` i `aca`.

$$\begin{cases} 240cca = 40aca + 360eca \\ 120eca = 20aca \end{cases}$$

Amb aquest sistema d'equacions es representa a la part esquerra els objectes que produeix la recepta en funció de la variable que indica la quantitat d'assemblers que la produeixen i a la part dreta la quantitat d'objectes que es necessiten en funció de la variable dels assemblers que necessiten l'objecte. D'aquesta manera s'assegura que per cada percentatge d'assemblers que produeixen una recepta hi ha exactament el nombre d'assemblers necessaris per poder abastir-lo.

Limitar l'espai de solucions

A banda del sistema d'equacions hem de delimitar quin és el nombre màxim d'*assemblers* que es poden usar, al cas del problema del *blueprint* serà el nombre màxim d'*assemblers* que es poden encabir en les dimensions del *blueprint* (*width*/3) × (*height*/3), en aquest exemple concret `max_assemblers = 9`. El problema que sorgeix amb la representació actual és que no tenim cap manera de determinar quin és el nombre d'assemblers individuals que s'utilitzen, ja que les variables `cca`, `eca` i `aca` són reals i si les sumem no obtenim el nombre d'assemblers, per això cal obtenir el ceil de cada variable real per poder imposar que la suma d'assemblers no pugui superar el nombre màxim de `max_assemblers`. El ceil de cada variable real es pot obtenir de dues maneres, la primera és una implementació que va molt lligada al funcionament del Z3 i la segona tracta d'una definició més formal del que significa el ceil.

La primera manera d'obtenir el ceil d'una variable real tracta de crear una variable auxiliar entera per cada variable real i definir una restricció que asseguri que l'unic valor que pot prendre la variable sigui el resultat d'aplicar el ceil a la variable real. Per a fer-ho es farà ús de la funció `ToInt()` del Z3 que rep per paràmetre la variable real i en retorna el valor enter, es a dir aplica un floor. Amb aquesta funció i la següent restricció podem definir el ceil:

LISTING 8.25: Ceil implementation 1

```

1 s = Solver()
2 x = Real("x")
3 x_ceil = Int("x_ceil")
4 s.add(IfToInt(x)<x, x_ceil==ToInt(x) + 1, x_ceil==ToInt(x)))

```

D'aquesta manera si el floor de la variable és més petit que el seu valor real significa que el seu ceil és el valor enter + 1 i en cas que el floor de la variable entera sigui igual que el valor real llavors el valor enter és el mateix que el valor real. Clarament, no cal tenir en compte el cas que el floor sigui més gran, ja que el floor d'un nombre decimal mai pot ser més gran que el mateix valor decimal.

Una altra manera d'obtenir el ceil d'una variable real tracta d'igual manera que amb l'anterior mètode crear una variable entera per cada real i definir una restricció on el valor que prengui la variable entera ha de ser més gran o igual a la real, ja que el ceil ha de ser un valor més gran en cas que la variable real tingui part decimal o el mateix valor en cas que la variable real no tingui part decimal. A banda per assegurar que el valor que prengui la variable entera no sigui més gran que el ceil de la variable real s'afegeix la restricció que fa que el valor de la variable entera menys 1 ha de ser més petit que el valor de la variable real d'aquesta manera si el valor enter és superior al ceil aquesta restricció no se satisfarà.

LISTING 8.26: Ceil implementation 2

```

1 s = Solver()
2 x = Real("x")
3 x.ceil = Int("x.ceil")
4 s.add(x.ceil-1<x)
5 s.add(x<=x.ceil)

```

Amb les variables que representen el ceil de cada variable només cal crear una restricció que asseguri que el nombre d'*assemblers* usats no superi el nombre màxim:

LISTING 8.27: Limit assemblers

```

1 s.add(ceil_cca+ceil_eca+ceil_aca<=9)

```

Optimització

Sense criteri d'optimització el solver trobarà qualsevol assignació que compleixi les restriccions imposades, però això no és el que volem, el nostre objectiu és saber quina recepta ha de tenir associada cada assembler tenint en compte que volem maximitzar la producció de l'objecte de sortida a l'entrada del model. Aquest pas va dur alguns problemes a l'hora de desenvolupar-lo, ja que l'optimitzador de Z3 permet maximitzar variables reals, la qual cosa pot semblar una tasca impossible perquè una variable real té infinites parts decimals dins un domini definit així que quan es va intentar optimitzar hi havia solucions subòptimes. Al final el que maximitzar la part entera més la part decimal, ja que si només tenim en compte la part entera hi ha múltiples assignacions a la part decimal que donen al mateix resultat a la part entera que podrien desencadenar en què les receptes de les quals depèn l'objecte objectiu no funcionessin al màxim rendiment.

LISTING 8.28: Optimization

```

1 s.maximize(aca.ceil + aca)

```

I la solució final és [aca.ceil = 5, eca.ceil = 1, cca.ceil = 3] i [aca = 5, eca = 5/6, cca = 25/12], així que ja sabem que per un blueprint on entren 9 *assemblers* i es vol produir advanced circuits, com a molt es necessitaran 5 *assemblers* dedicats a fabricar advanced circuits, 3 a fabricar copper cable i 1 dedicat a electronic circuits.

8.3.4 Eliminació de simetries

Tot i que l'optimització anterior elimina les simetries on dos o més *assemblers* amb diferents receptes associades es poden permutar obtenint el mateix resultat, en cas

que la recepta sigui la mateixa la simetria persisteix.

Per eliminar-la hem d'imposar ordre entre els *assemblers* que comparteixen la mateixa recepta d'aquesta manera les assignacions que siguin simètriques entre elles només seran satisfables si els *assemblers* tenen cert ordre.

La implementació actual no permet ordenar directament les posicions dels *assemblers*, ja que aquests no tenen una variable posició. Per poder ordenar-los s'han creat dues variables auxiliars *x* amb domini $[0..height]$ i *y* amb domini $[0..width]$ per tots els *assemblers* que comparteixen la mateixa recepta, això ho podem fer gràcies a l'optimització anterior que precalcula quina recepta ha de produir cada *assembler*.

Amb aquesta variable no és suficient, ja que d'alguna manera hem d'enllaçar les posicions ordenades amb la variable *assembler*, aquí sorgeix un altre problema, ja que el pre càlcul de les receptes calcula el màxim d'*assemblers* de cada recepta que es podrien arribar a usar, però no tenen per què usar-se tots, així que d'alguna manera cal assegurar-se que es faci l'ordenació de posicions i l'enllaç amb una quantitat no fixada d'*assemblers*. Per resoldre aquest problema s'ha afegit una tercera variable booleana *used* que indica si l'*assembler* s'està usant. D'aquesta manera amb una simple implicació lògica es pot forçar que només els *assemblers* en ús s'enllacin i s'ordenin.

Les restriccions que asseguren l'ordenació i l'enllaç són força simples. Per l'ordenació només cal que per cada parella de posicions x_1x_2 i y_1y_2 dels *assemblers* que comparteixin la mateixa recepta la posició $x_1 \leq x_2$ ja que en tractar-se d'un tauler 2D és possible que les coordenades *x* coincideixin, en aquest cas també cal assegurar que $y_1 < y_2$, per les coordenades *y* no cal tenir en compte el cas en què siguin iguals, ja que això significaria que els dos *assemblers* se superposen, cosa que mai pot passar.

LISTING 8.29: Posicions ordenades

```

1 for recipe in assembler_recipes:
2     assemblers = assembler_recipes[recipe]
3     assembler_keys = list(assemblers.keys())
4     for i in range(len(assembler_keys) - 1):
5         x0, y0, used0 = assemblers[assembler_keys[i]]
6         x1, y1, used1 = assemblers[assembler_keys[i + 1]]
7         symmetry_breaking.append(UGE(used0, used1))
8         symmetry_breaking.append(Implies(And(used0 == 1, used1 == 1), And(
    ULE(x0, x1), Implies(x0 == x1, ULT(y0, y1)))))


```

En aquest fragment de codi la variable *assembler_recipes* és un diccionari on les claus són les receptes que poden requerir més d'un *assembler* i el valor tracta d'un altre diccionari indexat per nombre d'*assembler* amb una llista que conté les variables de posició i la variable booleana anteriorment esmentades. Gràcies a poder guardar les variables Z3 en estructures Python la implementació de la restricció es torna molt senzilla, fent que només s'hagi d'iterar per cada recepta i per cada *assembler* que usa la recepta i assegurar l'ordre de les posicions. A més també es pot veure que com a petita optimització també s'ordenen les variables booleanes per així forçar que els *assemblers* s'usin en ordre evitant més simetries degudes a la possible combinació d'*assemblers* amb la mateixa recepta.

L'enllaç de les posicions *x* i *y* amb la variable *assembler*, es fa amb una restricció que per cada posició de la variable *assembler* si aquesta coincideix amb el valor *x*

y pres per algun dels *assemblers* i a més la seva variable booleana `used` és 1 llavors, el valor pres a la posició x y de la variable `assembler` ha de ser la del *assembler* representat per la variable x y .

Amb aquest enllaç, però no és suficient, ja que és unidireccional, és a dir només assegura que les variables ordenades siguin presents a la variable `assembler`, però no assegura que els valors presos a la variable `assembler` siguin presents a les posicions ordenades. Es necessita un segon enllaç en la direcció oposada, aquest es farà senzill, el que es fa és que per cada posició de la variable `assembler` si el valor pres coincideix amb algun *assembler* representat per les variables x y llavors la variable `used` haurà de ser 1, creant així el doble enllaç.

La restricció és una mica enrevessada, però amb l'exemple de la implementació s'enfent molt millor.

LISTING 8.30: Enllaç entre variables

```

1 for i in range(placement_height):
2     for j in range(placement_width):
3         for recipe in assembler_recipes:
4             assemblers = assembler_recipes[recipe]
5             assembler_keys = list(assemblers.keys())
6             for k in range(len(assembler_keys)):
7                 x, y, used = assemblers[assembler_keys[k]]
8                 symmetry_breaking.append(Implies(assembler[i][j] ==
assembler_keys[k], used == 1))
9                 symmetry_breaking.append(Implies(And(x == i, y == j, used
== 1), assembler[i][j] == assembler_keys[k]))

```

Les dues últimes línies són les que creen els enllaços en les dues direccions.

Capítol 9

Disseny del front end

Apèndix A

Frequently Asked Questions

A.1 How do I change the colors of links?

The color of links can be changed to your liking using:

```
\hypersetup{urlcolor=red}, or  
\hypersetup{citecolor=green}, or  
\hypersetup{allcolor=blue}.
```

If you want to completely hide the links, you can use:

```
\hypersetup{allcolors= . }, or even better:  
\hypersetup{hidelinks}.
```

If you want to have obvious links in the PDF but not the printed text, use:

```
\hypersetup{colorlinks=false}.
```

Bibliografia

- [1] Wube Software. *Factorio*. 2015. URL: <https://www.factorio.com/>.
- [2] Philip Zucker. *Z3 walk through*. 2019. URL: https://colab.research.google.com/github/philstook58/z3_tutorial/blob/master/Z3%20Tutorial.ipynb.
- [3] Nikolaj Bjørner Leonardo de Moura Lev Nachmanson i Christoph Wintersteiger. *Programming Z3*. URL: <https://theory.stanford.edu/~nikolaj/programmingz3.html>.
- [4] Sean Patterson Joan Espasa Mun See Chang Ruth Hoffmann. *Towards Automatic Design of Factorio Blueprints*. 2023. arXiv: 2310.01505 [Artificial Intelligence].
- [5] Erika Ábrahám. *SAT and SMT solving*. URL: <https://ths.rwth-aachen.de/wp-content/uploads/sites/4/teaching/arc-teaching-material/smt-short.pdf>.
- [6] *Z3 prover repository*. URL: <https://github.com/z3prover/z3>.