Informe de la Pràctica 2

Roger Baiges Trilla, Pau Prat Moreno i Cai Selvas Sala

5 de desembre de 2023

Universitat Politècnica de Catalunya

Grau en Intel·ligència Artificial

Algorismes Bàsics per la Intel·ligència Artificial

Resum

Aquest document correspon a l'informe de la segona pràctica de l'assignatura Algorismes Bàsics de la Intel·ligència Artificial del Grau en Intel·ligència Artificial de la Universitat Politècnica de Catalunya (UPC).

Al llarg d'aquest informe, s'esmenta el plantejament realitzat per tal de resoldre el problema, els passos que hem seguit, l'estructura i implementació del codi en PDDL i Python, els jocs de prova utilitzats, els experiments realitzats, els resultats obtinguts i les conclusions de la pràctica.

Índex

1 Introducció

Aquesta pràctica de planificació aborda un problema contemporani i rellevant: la creació d'un pla de lectura personalitzat. Pels amants de la lectura, aquesta pot ser una eina molt útil per tal d'ajudar als lectors a organitzar-se el temps de lectura de manera eficaç. El repte principal és desenvolupar un sistema que no només recomani llibres basant-se en les preferències dels usuaris, sinó que també tingui en compte l'ordre necessari per a una comprensió òptima de les sagues o sèries, incloent els llibres predecessors i paral·lels en l'ordre adequat.

Aquesta tasca s'aborda mitjançant el modelatge de dominis i problemes utilitzant el llenguatge PDDL (Planning Domain Definition Language). El problema es modela com un conjunt de llibres amb diferents relacions; per una banda hi ha predecessors, en què un conjunt de llibres pot representar l'eix cronològic d'una història, i per tant, hi ha llibres que cal llegir abans d'altres per comprendre la trama. Per altra banda també hi ha llibres paral·lels, que són llibres que ocorren simultàniament dins d'un mateix univers narratiu i que són preferibles llegir en un ordre específic.

El desenvolupament de la solució es divideix en diversos nivells, cadascun representant una complexitat creixent i una major profunditat en el modelatge del problema. Aquests nivells van des de la creació d'un pla bàsic de lectura, sense llibres paral·lels i amb un únic predecesor, fins a versions més avançades que incorporin un nombre variable de predecesors i llibres paral·lels, així com una restricció en el nombre de pàgines que es poden llegir per mes.

Un aspecte important i a tenir en compte d'aquesta pràctica és la generació de jocs de proves i l'avaluació empírica dels resultats. Això inclou la creació d'un programa per generar problemes de prova de manera aleatòria, el qual permet a l'usuari analitzar i validar la seva solució en un ampli rang de casos. A més, realitzarem una experimentació rigorosa per analitzar l'impacte del nombre de llibres i les seves dependències en el rendiment del planificador, oferint una oportunitat valuosa per tal de connectar la teoria impartida a l'assignatura amb la pràctica, i d'aquesta manera aprofundir en la comprensió dels sistemes basats en el coneixement.

En resum, aquesta pràctica ofereix una bona oportunitat per tal d'explorar el modelatge de problemes de planificació en l'àmbit de la creació d'un pla de lectura personalitzat, on destaca la importància de la generació de jocs de proves i la posterior avaluació empírica, que proporciona una connexió pràctica amb la teoria que hem estudiat a l'assignatura. A través d'aquesta pràctica, busquem aprofundir en la comprensió d'aquest nou llenguatge per a nosaltres (pddl) i la seva utilitat en el context ja esmentat.

1.1 Descripció del problema

L'enunciat del problema demana el desenvolupament d'un planificador de lectura de llibres durant els mesos d'un any però no esmenta enlloc com ho ha de realitzar ni en quin ordre. Per tal de fer un planficador que es podria utilitzar en situacions reals vam decidir que en el nostre programa les planificacions començarien al gener i anirien canviant al mes consecutiu a mesura que fes falta, degut al límit de pàgines o a les condicions causades per les relacions entre els propis llibres. També vam decidir que el planficador intenti realitzar la lectura en el mínim de mesos possibles per tal de que si l'usuari es vol llegir pocs llibres no faci falta esparar molts mesos.

Aquesta implementació no només té uns resultats mes fàcils d'interpretar ja que no es va saltant de mes a mes de forma aleatòria i no cronològicament, sinó que retalla enormement l'espai de cerca ja que per cada llibre només l'intentarà llegir en el mes actual i no haurà de provar altres combinacions resultant en un codi menys eficient.

Cal remarcar que totes les assumpcions que hem fet de l'enunciat i la nostra visió sobre com hem enfocat la pràctica està explicat detalladament al llarg de l'informe, en què per a cada secció, detallem quina interpretació hem fet de l'enunciat i com hem dut a terme la seva realització.

2 Documents i estructura general de l'entrega

Aquesta entrega, tal i com es requeria, s'ha fet a través d'un sol arxiu comprimit `.zip`. En l'interior d'aquest arxiu es troba l'aquest informe (en format *pdf*) i un altre arxiu comprimit `.zip` on hi ha tots el arxius amb codi (tant PDDL com Python) i les dades de les execucions dels experiments. Aquesta estructura no era exactament la que es demanava per l'entrega, però és la necessària (i creiem que més ordenada) donada la complexitat del nostre generador de jocs de prova que s'explicarà més endavant.

A continuació s'explica el contingut del segon arxiu comprimit `.zip`. És molt important **no modificar la jerarquia de directoris ni cap dels seus arxius** per tal de que l'execució dels arxius Python es pugui realitzar correctament.

- **Directori 'domains'**: en aquest directori es troben tots els arxius PDDL amb els dominis per cada un dels nivells d'extensió requerits en l'enunciat (Bàsic, 1, 2 i 3). Cada un d'aquests arxius s'anomena `default_domain_ext_X.pddl`, on `X` és el nivell d'extensió corresponent al domini.
- **Directori 'executables'**: aquest directori conté dos subdirectoris amb el següent contingut:
  - **Subdirectori 'macos'**: conté tots els arxius necessaris per compilar i posteriorment executar el programa *ff*. Aquests arxius són vitals per poder realitzar l'execució automàtica dels jocs de prova en un sistema operatiu *MacOS* i permeten que l'usuari no hagi te tenir el programa prèviament instal·lat. Si hi ha algun error en la compilació del programa, les execucions dels jocs de prova s'hauran de realitzar manualment.
  - **Subdirectori 'windows'**: conté l'executable i l'arxiu necessari per tal de poder executar el programa *metricff* en sistemes operatius *Windows*. Aquests arxius són vitals per l'execució automàtica dels joc de proves i permeten que l'usuari no hagi hagut d'instal·lar prèviament l'executable. Si hi ha algun error amb l'executable, les execucions dels jocs de prova s'hauran de realitzar manualment.

    Cal mencionar que no hi ha subdirectori per *Linux* perquè, degut a que no tenim cap integrant del grup amb sistema operatiu *Linux*, no hem volgut implementar la possibilitat d'executar automàticament els jocs de prova en aquest sistema operatiu davant del risc de no poder validar el seu funcionament.
- **Directori 'experiments'**: aquest directori conté dos subdirectoris amb el següent contingut:
  - **Subdirectori 'data'**: conté totes les bases de dades (arxius `.csv`) amb la informació de les execucions de cada un dels experiments realitzats. Tots els arxius s'anomenen `experimentX.csv`, on `X` és el nombre de l'experiment corresponent a la base de dades de l'arxiu en particular.
  - **Subdirectori 'plots'**: conté les imatges en format `.png` corresponents a les gràfiques (o *plots*) generades amb les dades dels diferents experiments (i que es troben, com ja hem mencionat, en el subdirectori 'data' del directori 'experiments').
  - **Subdirectori 'code'**: guarda el següent contingut:
    - **Subsubdirectori 'experiment2'**: conté tots els jocs de prova PDDL utilitzats en l'experiment 2.

  - **Subsubdirectori 'experiment3**: conté tots els jocs de prova PDDL utilitzats en l'experiment 3.

    Els jocs de prova utilitzats en l'experiment 1 no es guarden en aquest directori, ja que es generen a través del generador de jocs de prova aleatoris i es poden replicar utilitzant els mateixos paràmetres i llavors, especificats (*seeed*) que es poden trobar en l'arxiu `experiment1.csv` del subdirectori 'data' del directori 'experiments'.

- **Directori 'problems'**: en aquest directori es troben tots els arxius amb els problemes (o jocs de prova) PDDL, tant els que hem creat manualment com els que es generen amb el generador aleatòri de jocs de prova (aquests últims es guarden en el directori quan es fan execucions; en la entrega inicial només hi ha els problemes generats manualmen perquè encara no s'ha fet cap execució del generador automàtic). Els arxius generats manualmen s'anomenen `default_problem_ext_X-vY.pddl`, on `X` és el nombre corresponent al nivell d'extensió del joc de proves i `Y` identifica la versió del joc de proves (1 o 2). Per altra banda, els arxius que es generin manualment s'anomenen `generated_problem_ext_X_vY.pddl`, on `X` és el nombre corresponent al nivell d'extensió del joc de proves i `Y` identifica la versió del joc de proves generat (n'hi haurà tants com l'usuari desitgi). Cal mencionar que cada vegada que s'executi de nou el generador i es trii un nivell d'extensió del joc de proves que hagi estat generat prèviament, els arxius es reemplaçaran en el directori (ja que s'anomenaran igual que els que hi havia prèviament).

- **Directori 'results'**: aquest directori conté tots els arxius `.txt` amb els resultats (plans de lectura) de tots els jocs de prova, tant els generats manualment com els generats (i posteriorment executats) pel generador aleatori de jocs de prova. Pel que fa als resultats dels jocs de prova generats manualment (és a dir, la "traça" dels jocs de prova generats manualment, que es requeria en l'entrega) tots s'anomenen `default_result_ext_X_vY.txt`, on `X` és el nombre corresponent al nivell d'extensió del joc de proves i `Y` identifica la versió (1 o 2) del joc de proves generat manualment (que es pot trobar, com ja hem mencionat, en el directori 'problems'). Cal mencionar que tots aquests resultats han estat obtinguts fent execucions sense l'optimitzador de mesos/passos (flag `-O`). Per altra banda, els resultats dels jocs de prova que es guardin en aquest directori després d'executar automàticament els jocs de prova generats pel generador aleatori, s'anomenaran sempre `generated_result_ext_X_vY.txt` o `generated_opt_result_ext_X_vY.txt` (on `X` és el nombre corresponent al nivell d'extensió del joc de proves i `Y` identifica la versió del joc de proves generat) en funció de si s'executen amb l'optimitzador de mesos/passo (flag `-O`) o no. En aquest directori, de la mateixa manera que passava en el directori 'problems', quan s'executin i guardin nous resultats es reemplaçaran els existents (generats en anteriors execucions del mateix nivell d'extensió).

- **Arxiu 'generar_jocs_proves.py'**: aquest arxiu Python conté tot el codi per poder executar el generador de jocs de proves aleatoris que s'explica en l'apartat 3.4.2 d'aquest document. És molt important **no canviar d'ubicació d'aquest arxiu** en cap moment per evitar que en la seva execució, donat que crea/accedeix a arxius, perdi el *path* d'alguns fitxers i no s'executi correctament. Per tant, l'execució d'aquest arxiu Python s'ha de fer des del directori principal ('ABIA_Practica2_Roger_Cai_Pau').

  També és important advertir que aquest arxiu no funcionarà correctament si no es tenen instal·lades les llibreries `networkx`, `numpy` i `matplotlib` no es podrà executar l'arxiu. Ad-

dicionalment, volem mencionar que els arxius han estat generats i provats amb les versions següents de Python i de les llibreries:

- `Python: 3.11.2`.
- `numpy: 1.26.0`.
- `networkx: 3.0`.
- `matplotlib: 3.8.0`.

Per tant, no podem garantir el funcionament correcte del programa en qualsevol execució realitzada amb versions diferents a aquestes.

- **Arxiu 'generar_plots_expermients.py**: aquest arxiu Python conté el codi per poder generar les gràfiques de cada un dels experiments. En executar-lo, es pregunta a l'usuari quin experiment vol realitzar (i amb quin nivell d'extensió, en cas que aquest sigui modificable en l'experiment escollit) i genera les gràfiques (que es poden trobar igualment en el subdirectori 'plots' del directori 'experiments'). És molt important **no canviar d'ubicació d'aquest arxiu** en cap moment per evitar que en la seva execució, donat que crea/accedeix a arxius, perdi el *path* d'alguns fitxers i no s'executi correctament. Per tant, l'execució d'aquest arxiu Python s'ha de fer des del directori principal ('ABIA_Practica2_Roger_Cai_Pau').

  Finalment, volem remarcar que aquest arxiu no funcionarà correctament si no es tenen instal·lades les llibreries `pandas`, `matplotlib` i `seaborn`. Concretament, les versions, tant de Python com de les llibreries, amb les que s'ha desenvolupat i provat el codi són les següents:

  - `Python: 3.11.2`.
  - `pandas: 2.1.0`.
  - `matplotlib: 3.8.0`.
  - `seaborn: 0.12.2`.

  Per tant, no podem garantir el funcionament correcte del programa en qualsevol execució realitzada amb versions diferents a aquestes.

3 Estructura i implementació del programa

En aquest apartat expliquem detalladament com hem modelat el domini del problema (variables, predicats i accions), com hem modelats els problemes a resoldre (objectes, estat inicial i estat final), així com la manera de com hem desenvolupat els diferents models.
També entrem en detall en com hem implementat els jocs de prova i què fa l'arxiu python, a més d'explicar la manera en com s'utilitza.
Primer, però, hem d'entendre quines són les característiques del nostre domini i com hem modelat aquest per tal que s'adeqüi al problema.

3.1 Característiques del domini

El domini del nostre problema està definit en el fitxer *default_domain_ext3.pddl*, que conté les variables, predicats i accions necessàries per tal de realitzar correctament el problema en qüestió. Aquest fitxer consta de certs requeriments, que es tracta d'una sèrie de característiques necessàries del llenguatge PDDL que es fan servir en la definició del problema, i en el nostre cas hem necessitat els següents:

- strips: llenguatge bàsic.
- typing: permet la definició de tipus en variables i constants.
- negative-preconditions: indica que es permeten precondicions negatives al definir les accions.
- disjunctive-preconditions: permet condicions de precondició disjuntes, és a dir, una acció pot tenir múltiples conjunts de precondicions i s'executarà si almenys una d'aquestes és certa.
- existential-preconditions: es permeten precondicions existencials. Això significa que una acció pot requerir que almenys un dels elements que compleixi una condició existeixi en l'estat actual.
- universal-preconditions: permet precondicions universals. Una acció amb precondicions universals només s'executarà si la condició és certa per a tots els elements especificats.
- fluents: permet utilitzar funcions numèriques, el valor de les quals pot anar canviant.

D'aquesta manera ja podem definir les variables corresponents al domini del problema.

3.1.1 Variables

Les variables juguen un paper fonamental en la representació de l'estat, ja que s'utilitzen com a identificadors genèrics dins dels predicats, funcions i accions del domini. Aquestes variables són placeholders que es substitueixen per objectes concrets quan es genera un pla a partir del problema. Cal remarcar que cada variable ha d'estar associada amb un tipus definit dins de `:types`, i en el nostre cas tenim dos objectes definits (com expliquem posteriorment amb més detall en l'apartat d'Objectes, veure 3.2.1), que són `book` i `month`. Per tant, sempre que aparegui una variable estarà representant, o bé un llibre, o bé un mes.
El primer lloc on associem les variables a objectes és en l'apartat de Predicats (veure 3.1.2), en què cada predicat utilitza variables per expressar les propietats dels objectes o les relacions entre ells. Per exemple, `(read ?b - book)` utilitza la variable `?b` per a representar qualsevol llibre que hagi estat llegit per l'usuari. Com que la variable `?b` ha estat tipificada com a `book`, això ens assegura

que només s'aplicarà a objectes del tipus llibre. Per tant, permetrà la reutilització del predicat per a múltiples llibres en diferents instàncies del problema.
En l'apartat de `:functions` (de l'arxiu de domini) també hi trobem un ús de variables, més concretament, quan fem servir `pages_read` i `total_pages`. En el primer cas, com que declarem la sentència `(pages_read ?m - month)`, sempre que cridem a aquesta funció, estarem fent referència a les pàgines llegides de un cert mes, ja que hem definit que la funció sempre retornarà un valor numèric referenta un mes (ja que l'objecte que hem associatt a la variable ?m és `month`). En l'altre cas, al ser la sentència `(total_pages ?b - book)`, aquesta sempre farà referència al nombre de pàgines d'un llibre, ja que en aquest cas hem associat la variable a l'objecte de tipus `book`.
Per últim, també és important recalcar l'ús de variables en les accions, tot i que entrem més en detall en l'apartat corresponent (veure 3.1.3).

3.1.2 Predicats

Els predicats del domini representen el problema, i tal com hem explicat anteriorment, per tal de representar un cert estat necessitem poder definir quins llibres han estat llegits, quins llibres son paral·lels o predecessors de quins, en quin mes s'ha de llegir cada llibre i quins son els llibres que l'usuari vol llegir, així com saber en quin mes ens trobem, per tal que el planificador pugui saber si ja no és possible assignar més llibres al mes actual (perquè excediríem el límit de 800 pàgines llegies mensuals) i així començar a assignar llibres al següent mes.
Aquests predicats són essencials perquè defineixen les propietats bàsiques dels objectes i les relacions entre aquests. Permeten al planificador entendre quin és l'estat actual i quin és l'objectiu a assolir. Com hem vist en l'apartat anterior, els predicats tenen molta importància en el domini ja que s'utilitzen també per a definir les precondicions i els efectes de les accions, descrivint així l'estat dels objectes (per exemple, si un llibre en particular ha estat llegit per l'usuari o no), les relacions entre objectes (com ara si dos llibres són paral·lels o si un cert llibre és predecessor d'un altre llibre). Amb això en ment, vam decidir que els predicats adients per expressar-ho correctament en PDDL serien els següents:

- **(read ?b - book)**
  Representa que el llibre *?b* ha estat llegit per l'usuari.
- **(predecessor ?b1 - book ?b2 - book)**
  Indica que el llibre *?b1* és predecessor del llibre *?b2*.
- **(parallel ?b1 - book ?b2 - book)**
  Indica que el llibre *?b1* és paral·lel al llibre *?b2*.
- **(goal_book ?b - book)**
  Representa que el llibre *?b1* és un llibre que l'usuari vol llegir.
- (**assigned ?b - book ?m - month**)
  Fa referència al mes en què es llegirà cada llibre. Per tant, cada llibre (*?b*) s'ha d'assignar a un mes en concret (*?m*).
- **(current_month ?m - month)**
  Representa el mes actual (*?m*).
- **(next_month ?m1 - month ?m2 - month)**
  Per a un mes *?m1*, indica que el següent mes és *?m2*.

- **(previous_month ?m - month)**
  Representa el mes anterior a l'actual (*?m*).

Ara, amb tots aquests predicats ja definits, ja tenim les eines necessàries per a representar qualsevol estat del problema, és a dir, podríem representar qualsevol situació diferent per a un determinat usuari.
Aquest conjunt de predicats permeten crear un model prou detallat del pla de lectura, on no només es pot planificar quins llibres llegir i en quin ordre, sinó també garantir que l'usuari assoleixi els seus objectius de lectura de manera eficient i satisfactòria. Per tant, ja tenim totes les eines per a descriure un cert estat, i ara necessitem poder operar sobre un estat per tal de transformar-lo i apropar-nos així cada cop més a l'objectiu fins tal d'assolir-lo. Això ho fan possible les diferents accions del domini, que són les que permeten modelar cada estat i transformar-lo per tal d'apropar-nos fins a la solució.

3.1.3 Accions

Després d'analitzar detingudament quines accions ens permetien realitzar el problema correctament i arribar a una solució, hem conclòs que amb tan sols dues accions ja es pot resoldre el problema: una que assigni un cert llibre a un mes en concret, i una altra que "finalitzi"el mes actual i comenci un mes nou, per tal que es puguin assignar llibres a tots els mesos de l'any.
Com es pot veure en l'arxiu de domini, cada acció consta d'uns paràmetres, unes precondicions i l'efecte que causa quan és realitzada.
Els paràmetres d'aquestes accions són variables que es defineixen per tal de representar els objectes sobre els quals l'acció pot operar. És a dir, serveixen per a especificar el tipus d'objectes que són necessaris a l'hora d'executar l'acció. D'aquesta manera, permeten que les accions siguin genèriques i es puguin utilitzar amb diferents instàncies d'objectes dels tipus corresponents. Per altra banda, les precondicions són afirmacions que han de ser certes per tal que l'acció pugui ser realitzada. Per tant, asseguren que l'acció només es dona quan té sentit lògic, evitant errors i ajudant a dirigir la cerca del planificadodr cap a l'objectiu. Finalment, els efectes de cada acció descriuen com canvia l'estat del problema després de realitzar l'acció en qüestió, és a dir, com canvien els predicats de cada llibre i mes en aplicar una certa acció, la qual cosa permet apropar-nos cada cop més cap a l'objectiu.
Així doncs, un cop entès la importància de les accions, el nostre domini consta de les següents:

- assign_book
Aquesta acció permet assignar un llibre a un mes en concret, per tant, es tracta de l'operador que provoca que l'usuari vagi llegint llibres i aquests s'assignin a diferents mesos, tenint en compte les restriccions del màxim de pàgines llegies mensualment o les condicions que han de complir els llibres que són predecessors o paral·lels dels llibres que l'usuari vol llegir.

- Paràmetres: `?b - book ?currentm - month ?prevm - month`, que representen, respectivament, el llibre que volem assignar, el mes actual, que és en el que assignarem el llibre si es compleixen les precondicions, i el mes anterior.

- Precondicions: En primer lloc, es verifica que el llibre no hagi estat llegit per l'usuari anteriorment, ja que si ha estat llegit, no l'hauríem d'assignar a cap mes pel fet que no l'hem d'incloure al pla de lectura.
  També es comprova que el llibre en qüestió compleixi una de les següents condicions: és un llibre que l'usuari vol llegir, és predecessor d'un altre llibre o té algun llibre paral·lel.

En aquest apartat de precondicions també definim que el mes actual és *?currentm* i l'anterior és *?prevm*. Això serveix per tal de retallar moltíssim l'espai de cerca ja que si un llibre no és objectiu ni és predecessor o paral·lel d'un llibre que sí que ho és, el planificador no el tindria ni en compte. En cas que no ho tinguessim, no només seria menys eficient, sinó que es podria donar el cas que ens fes llegir un llibre que no és necessari degut a que no ho hem impedit.
A més, es constata que els predecessors d'aquest llibre han d'haver estat llegits en mesos anteriors, i a més, els llibres paral·lels han de ser llegits abans del mes del llibre *?b* o durant el mateix mes, seguint les condicions especificades a l'enunciat de la pràctica. Finalment, es comprova que la suma de les pàgines llegides durant el mes actual més les pàgines d'aquest llibre no superi 800, que és el límit de pàgines que l'usuari pot llegir mensualment, per tant, no s'assignarà cap llibre que faci excedir aquest límit en un cert mes.

- Efecte: Es marca el llibre com a llegit i s'assigna al mes actual. També s'incrementa la quantitat de pàgines llegides durant el mes actual, sumant-li el total de pàgines del llibre en qüestió.

- start_month
Aquesta acció ens permet anar avançant de mes per tal de poder assignar llibres a mesos nous, ja que si només comptéssim amb l'acció d'`assign_book`, un cop arribéssim al límit de 800 pàgines llegides mensuals i no poguéssim assignar més llibres en aquell mes, el planificador no arribaria a l'objectiu de fer que l'usuari es llegeixi tots els llibres que volia llegir inicialment junt amb els predecessors i parallels corresponents, ja que tots aquells llibres que fossin `goal_book` i que no haguessin estat assignats en el primer mes, quedarien sense ser assignats i, per tant, l'usuari mai arribaria a llegir-los.
És per això que, malgrat pugui semblar una acció en certa manera simple o bàsica, és indispensable que existeixi ja que d'aquesta manera el planificador pot anar assignant els llibres que l'usuari ha de llegir en els diferents mesos.

- Paràmetres: `?nextm - month ?currentm - month ?prevm - month`, que representen el pròxim mes, l'actual i l'anterior.

- Precondicions: Simplement es comprova que `?currentm` és el mes l'actual i `?prevm` és l'anterior. També s'especifica que el pròxim mes és `?nextm` gràcies al predicat `(next_month ?currentm ?nextm)`, que com hem vist anteriorment indica que `?currentm` i ?`nextm` són mesos consecutius.

- Efecte: Es finalitza el mes actual i comença un nou mes. Per tant, necessitem declarar que el mes actual ja no és `?currentm`, sinó `?nextm`, i el mes anterior a l'actual, lògicament, és `?currentm`.

Importància de les precondicions:

En el cas que no es compleixin les precondicions d'una certa acció, aquesta no es durà a terme, per lo qual el planificador buscarà de satisfer aquestes precondicions en el cas que sigui necessari aplicar una certa acció. Per exemple, si per a què l'usuari es llegeixi el llibre `book4` que, posem per cas, és un `goal_book`, primer es necessita que llegeixi el llibre `book3`, que és un predecessor, per tant el planificador farà que l'usuari primer llegeixi `book3` i, en mesos posteriors, `book4`, ja que si no ho fa, les precondicions mai es complirien per a assignar `book4` a un mes, i per tant, el problema mai es resoldria ja que l'usuari no arribaria a llegir un llibre que és `goal_book` (en aquest cas, `book4`). És a dir, l'apartat de precondicions és molt important i és el que li dóna la consistència lògica al

pla de lectura, ja que assegura que les accions només es duen a terme quan tenen sentit dins del context del problema.
A més, no només això, sinó que també fan que el programa sigui molt més eficient, ja que serveixen com a "filtre"per tal que el planificador pugui descartar ràpidament les accions que no són aplicables en un determinat estat, millorant així l'eficiència del procés de planificació. Sense aquestes restriccions, el planificador hauria de considerar un nombre molt més gran de possibles accions en cada pas, incrementant d'aquesta manera el temps d'execució del programa i fent que aquest fos molt més ineficient.
Per tant, les precondicions ajuden a definir un camí cap a l'objectiu, marcant les etapes que cal assolir abans d'arribar a una solució, evitant així que s'executin accions innecessàries, i d'aquesta manera cada acció hagi de contribuir a aconseguir l'objectiu final.

3.2 Característiques del problema

3.2.1 Objectes

Només hem necessitat definir dos objectes; llibre i mes, que representen un llibre qualsevol del catàleg i un dels dotze mesos de l'any (gener, febrer, març, etc.), al qual se li assignaran un cert nombre de llibres, seguint les restriccions de no excedir les 800 pàgines llegides. Així doncs, els objectes que haurem de definir en el nostre arxiu de problemes seran els diferents llibres del catàleg i els dotze mesos diferent, per tal que cada llibre del pla de lectura es pugui assignar a un mes ja inicialitzat. Com veurem posteriorment en l'apartat de Jocs de Prova (3.4), haurem d'assignar-los tal que així:
`book1 book2 book3 book4 ... book15 - book`
`Past January February March ... December - month` (en què Past representa tots el mesos anteriors a January, és a dir, si un llibre s'assigna a Past, vol dir que l'usuari va llegir aquest llibre en qüestió en un passat).
En aquest exemple hem suposat que tenim 15 llibres al catàleg, però lògicament haurem de definir tots els llibres que tinguem, és a dir, si tenim 28 llibres, haurem d'expressar que tots aquests llibres pertanyen a l'objecte `book`.

3.2.2 Estat inicial

L'estat inicial representa la descripció de les condicions inicial abans de fer qualsevol acció, és a dir, la configuració a l'inici del problema, incloent aspectes com els llibres que l'usuari ja ha llegit, els llibres que vol llegir, i qualsevol altra informació rellevant que defineixi la situació de partida des de la qual partirà el planificador.
Així doncs, per tal de crear l'estat inicial necessitem dur a terme el següent:

Definir quins llibres són predecessors de quins:

Necessitem saber quins són els llibres que són predecessors d'altres llibres, lo qual ho fem utilitzant el predicat `(predecessor book1 book2)`, en què book1 serà predecessor de book2.
Si per exemple tenim una saga de llibres que consta de 4 llibres diferents cadascun és predecessor del següent, és a dir, la història transcorre cronològicament, tenint en compte que els llibres es diuen llibre_saga1, llibre_saga2, llibre_saga3 i llibre_saga4, haurem de declarar els següents predicats:
`(predecessor llibre_saga1 llibre_saga2)`
`(predecessor llibre_saga2 llibre_saga3)`

`(predecessor llibre_saga3 llibre_saga4)`

Definir els llibres que són paral·lels:

Com que també hem de saber quins llibres són paral·lels entre sí, ho haurem de declarar mitjançant el predicat `(parallel book1 book2)`, per definir que book1 i book2 són paral·lels. Cal remarcar que per tal de no tenir problemes amb el codi i optimitzar-lo, si el llibre *book7* té, per exemple, tres llibres paral·lels (*book3*, *book5* i *book8*), haurem de definir els predicats de manera que el llibre *book7* sigui el primer paràmetre, és a dir, farem sevir el predicat `(parallel book7 book3)`, i el mateix per al llibre 5 i llibre 8. D'aquesta manera, ens fixem que reduïm les línies de codi de manera significativa, ja que si féssim totes les parelles de llibres paral·lels, hauríem d'escriure n sobre 2 combinacions, mentre que en aquest cas únicament hem de fer-ho (n-1) cops. En l'exemple anterior, podem veure que obviem el fet de definir `(parallel book3 book5)`, `(parallel book3 book8)`, `(parallel book5 book8)`, etc., ja que com que tots són paral·lels al llibre 7, també són paral·lels entre ells, per tant, ens serveix per optimitzar molt el codi.

Definir els llibres que l'usuari vol llegir:

Necessitem saber quins són els llibres objectius per tal de poder arribar a un estat final en què tots ells s'hagin llegit, junt amb els seus predecessors i paral·lels corresponents. Fem servir la sentència `(goal_book book7)` per determinar que book7 és un llibre que l'usuari vol llegir. Així doncs, ens assegurem que aquests llibres apareixeran sí o sí al pla de lectura, simplement ens falta assignar-los a un mes en concret, lo qual ja ho farà el planificador. Podem observar que, com més nombre de llibres*goal_book* creem en l'estat inicial, és a dir, com més llibres vulgui llegir l'usuari, més complexitat tindrà el programa, ja que també s'hauran de llegir els seus pertinents llibres paral·lels i predecessors seguint les restriccions imposades a l'enunciat.

Per cada llibre, saber quantes pàgines té:

Com que l'usuari no pot llegir més de 800 pàgines cada mes, hem de saber quantes pàgines té cada llibre per tal d'assginar-los ed la manera més eficient possible en els diferents mesos tenint en compte aquesta restricció. Així doncs, fent ús de la variable *total_pages*, inicialitzem el seu valor per a cada llibre, ja que d'aquesta manera podrem saber quantes pàgines ha llegit l'usuari cada mes i, d'aquesta manera, no sobrepassar les 800 pàgines llegides mensuals. Per exemple, si book1 té un total de 700 pàgines, necessitem fer ús de la sentència `(= (total_pages book1) 700)`. Cal remarcar també que no té sentit inicialitzar aquesta variable a un número n en què `n>800`, ja que aleshores el llibre en qüestió no s'assignaria a cap mes i, per tant, mai apareixeria al pla de lectura. Per altra banda, com expliquem en l'apartat posterior de Jocs de Prova (3.4), aquest número ha de ser `n>15`, ja que suposem que tots els llibres del catàleg tenen, com a mínim, 15 pàgines. Per tant, com menys pàgines tinguin els llibres que creem en l'estat inicial, menys trigarà l'usuari en llegir-los i, conseqüentment, menys mesos es crearan (minimitzant així el valor de la variable *num_months_created*).

Inicialitzar a 0 les pàgines llegides per cada mes:

Per tots els mesos que haguem inicialitzat en crear els objectes (excepte Past, que representa el passat), és a dir, de gener a desembre, necessitem definir que l'usuari encara no ha llegit cap pàgina, ja que en aquests mesos encara no ha llegit cap llibre. Per tant, sent *month* un mes qualsevol entre gener i desembre, hem de definir que el seu valor inicial és 0, i a mesura que anem assignant llibres al mes en qüestió, anirem incrementant el valor de la variable segons les pàgines que tingui cada

llibre, la qual cosa hem explicat amb més detalla en l'apartat d'Accions (3.1.3). Per fer-ho, hem de fer servir la sentència `(= (pages_read month) 0)` per cada mes diferent. D'aquesta manera aconseguim que en l'estat inicial es compleixi que l'usuari encara no ha llegit cap llibre durant els mesos als quals el planificador assignarà els llibres pertinents al pla de lectura.

Inicialitzar a 1 la variable *num_months_created*:

Com hem explicat anteriorment en l'apartat de Descripció del problema (1.1), sempre considerem que el primer mes del pla de lectura és el gener, i a partir d'aquí anem assignant llibres als mesos conseqüents. Tenint això en compte, com que comencem directament ja en el primer mes (gener), la variable *num_months_created*, que representa el número de mesos que s'han creat fins al moment, ha de prendre com a valor 1. Un cop finalitzi l'assignació de llibres del gener i passem a febrer, incrementarem el seu valor en una unitat, i farem això successivament fins que es compleixi l'objectiu del programa.
Així doncs, l'estat inicial sempre tindrà la variable *num_months_created* amb valor 1.

Declarar quin és el mes actual:

Necessitem definir en quin mes comença el pla de lectura, que com tot just hem comentat, serà el gener. Per tant, sempre haurem de fer ús del predicat `(current_month January)` en crear l'estat inicial a l'arxiu del problema.

Definir el passat com a previous_month:

Com que pot ser que l'usuari hagi llegit llibres en un passat, hem de declarar que "el mes anterior"a l'actual (gener) és *Past*. Per tant, també haurem d'utilitzar sempre el predicat `(previous_month Past)`.

Assignar els llibres ja llegits i marcar-los com a llegits:

En cas que l'usuari ja hagi llegit algun llibre del catàleg, definir quin ha estat i marcar-lo com a llegit, per tal de no incloure'l en el pla de lectura més endavant. A més, l'hem d'assignar al passat (*Past*), ja que només volem assignar als mesos els llibres que l'usuari encara no hagi llegit. Per exemple, si l'usuari ha llegit el llibre *book1*, ho hem d'expressar de la següent manera:

```
(assigned book1 Past)
(read book1)
```

3.2.3 Estat final

L'estat final representa l'objectiu del problema, que defineix les condicions sota les quals aquest es considera resolt, i per tant, serveix com a guia del planificador per tal de trobar la millor solució possible. En aquest problema, l'estat final representa l'estat en què s'han llegit tots els llibres que l'usuari volia llegir, i per tant, tots han estat assignats a uns mesos en concret, i en el cas que tinguessin predecessors que l'usuari encara no havia llegit, també s'hagin assignat de manera que l'usuari els llegeixi en un mes anterior al del llibre que volia llegir inicialment. A més, si el llibre que l'usuari vol llegir també té llibres paral·lels, aquests han d'haver estat llegits en el mateix mes que el llibre en qüesió o durant el mes anterior.
Això ho definim al mateix arxiu de problema, amb la condició de que tots els llibres que són `goal_book` han de ser llegits, és a dir, si *book5* és un llibre que l'usuari vol llegir (definit amb el predicat `(goal_book book5)`), aleshores s'ha de complir que aquest llibre estigui llegit, per tant,

s'ha de complir el predicat `(goal_book book5)`. Un cop es compleixi aquesta condició ja esmentada per a tots els llibres que l'usuari volia llegir, es considera que el problema ja està resolt.
Com hem dit anteriorment i com s'especificava en l'enunciat de la pràctica, per a considerar que el problema està resolt s'hauria de complir que tots els predecessors d'un *goal_book* han haver d'estat llegits en mesos anteriors, i també que tots els llibres paral·lels es llegeixen el maetix mes o el mes anterior. Tot i així, realment només cal posar com ha objectiu que els llibres que l'usuari vol llegir, ja els hagi llegit, ja que és la pròpia acció *assign_book* la que provoca, en l'apartat de precondicions, que s'hagi de llegir un cert predecessor d'un llibre abans d'aquest o què un llibre paral·lel s'assigni al mes actual o anterior del llibre objectiu (en el cas que no hagi estat llegit per l'usuari). Per tant, com que és la mateixa acció la que provoca que es compleixin les condicions, nosaltres simplement hem de comprovar que l'usuari hagi llegit els llibres que tenia com a objectiu inicialment. Això ho podem veure a l'apartat de *:goal*, en què sempre s'ha d'especificar quin és l'objectiu del problema, i en el nostre cas, la sentència és la següent:
`(forall (?b - book) (imply (goal_book ?b) (read ?b)))`
A banda d'això, per tal de poder trobar una millor solució, vam decidir minimitzar la variable *num_months_created*, ja que d'aquesta manera el planificador tracta de crear els mínims mesos possibles, la qual cosa fa que els llibres s'assignin als diferents mesos de manera més eficient, fent que l'usuari trigui menys temps en llegir tots els llibres del pla de lectura. Això ho vam implementar amb la sentència `(:metric minimize (num_months_created))`, que indica que *num_months_created* és la mètrica a minimitzar.

3.3 Desenvolupament dels models

Com que des d'un inici teníem clar com volíem implementar l'extensió 3 (quines variables fer servir, predicats, accions, etc.), el que vam decidir és fer primer aquesta extensió, i després ja trauríem el que calgués per a cada extensió, per tal de poder tenir un arxiu de domini i un de problema per a cada extensió diferent.

Com ja s'ha observat en els apartats anteriors, hem acabat creant 2 opeadors *assign_book* i *start_month* alhora que 9 predicats diferents i 3 fluents. Inicialment la primera pregunta va ser com es pot representar tot el domini amb el mínim número de *types*. El primer va ser evident, els llibres, però com que el nostre objectiu era un planificador d'un any mes per mes vam decidir que els tipus del nostre treball serien llibres i mesos i que qualsevol altres relació seria expressada en predicats. Una altra opció podia ser, per exemple, crear subtipus de llibre que fossin *paral·lel* i *predecessor* però vam arribar a la conclusió que això no ens permetria que un llibre fos paral·lel i predecessor d'un altre llibre alhora i, per tant vam acabar decidint-nos per la nostra implementació inicial.

Ja que el nostre objectiu principal era assolir primer l'extensió màxima per poder després crear les altres, vam començar plantejant els predicats. El primer va ser molt obvi, un llibre s'ha de poder llegir i en cas que ja s'ha llegit utilitzaríem `(read ?b)`. Una vegada vam crear aquest predicat vam veure que ens faria falta també un predicat per marcar si un llibre és objectiu o no ja que això ens ajudaria molt alhora de reduir les opcions als diferents operadors; el predicat es va dir `(goal ?b)`. Per tal de decidir si un llibre era predecessor o paral·lel d'un altre vam crear uns predicats que no semblaven gaire complicats `(predecessor ?b1 ?b2)` i `(parallel ?b1 ?b2)` però aquest últim vam veure que podríem optimitzar-lo molt si afegíssim restriccions. Si en una saga, per exemple, *Harry Potter* que consta de 7 llibres consecutius, per representar-ho necessitaríem tan sols 7 predicats `predecessor` però en canvi si fossin paral·lels i volguessim representar totes les relacions entre tots hauríem de declarar un total de $\binom{7}{2} = 21$ predicats totals sense tenir en compte el factor

recíproc de dos llibres paral·lels. Per tal de simplificar aquest cas vam decidir que es seleccionaria un llibre als quals tots els altres llibres para·lels serien paral·lels seus i així amb tan sols 7-1 = 6 predicats declarats podríem representar un grup de 7 llibres tots relacionats entre si. El que abans ens costava predicats de l'ordre quadràtic, degut a les combinacions sobre 2, ara tan sols lineal. A més a més ja vam fer això expressament degut a que abans de programar ja vam pensar l'algorisme per tal de que els llibres paral·lels es llegissin tots en un període de dos mesos i la manera més òptima que sens va acudir va ser que un llibre es guardés pel final i aquest controlés que tots els altres s'haguessin llegit o en el mes passat o en l'actual.

Ara tan sols faltava una manera de representar els mesos i una manera per poder guardar en quin mes s'havia llegit un llibre, per tal de poder realitzar les condicions sol·licitades en l'enunciat sobre que els llibres predecessors s'han d'haver llegit en un altres mes i que els paral·lels en un període de dos mesos. El predicat `(assigned ?b ?m)` es declara una vegada un llibre s'ha llegit (`read ?b`) i serveix per poder saber quan. Els altres predicats que ens farien falta per poder tractar amb els mesos serien `(current_month ?m)` per saber en quin mes es troba el planifcador `(previous_month)` per poder saber si un llibre paral·lel ha estat assignat en el mes passat i finalment `(next_month)` per indicar un ordre cronològic entre mesos (`(next_month gener febrer)`, `(next_month febrer març)`...). Ara ja disposàvem de totes les eines per tal de poder descriure un estat alhora que totes les relacions dels seus diferents elements.

Finalment ens feia falta la creació de les diferents accions per tal de poder planificar la lectura dels llibres objectius. Només volíem utilitzar un operador per llegir els llibres i una altra que s'encarregués de la gestió del temps i, per tant, dels mesos.

L'operador `(assign_book)` és, probablement, la part més important del nostre codi en pddl ja que s'encarrega d'assignar un llibre a un mes del pla de lectura. Com ja s'ha explicat al punt 3.1.3 com a paràmetres té un llibre, i dos mesos (l'actual i el previ). Inicialment només comptava amb les precondicions de que per cada llibre el seu predecessor s'ha d'haver llegit en un altre mes que no sigui l'actual; degut a la nostra simplificació del planificador on tot funciona cronològicament, si el llibre s'ha llegit i no en el mes actual significa que s'ha llegit en un mes anterior. A part d'això teníem la precondició dels llibres paral·lels que obligava a que tots els llibres paral·lels a aquest calien ser llegits o el mes anterior (el teniem com a paràmetre de l'operador) o en el mes actual. Com ja s'ha mencionat anteriorment, vam decidir fer que tots els grups de llibres paral·lels ho fossin tan sols a un llibre per tal de poder simplificar el nombre de predicats però també degut a que d'aquesta manera podem complir les condicions de l'enunciat de que caga grup de llibres paral·lels s'han de llegir en el període de 2 mesos consecutius fent que el llibre als quals tots són paral·lels sigui l'últim i els altres s'hagin llegit o en aquell mes o en l'anterior. Aquesta lògica fa que el nostre codi sigui molt més òptim i eficient ja que no ha de tenir en compte pràcticament res a banda dels dos mesos ja mencionats. Per tal d'assolir la última extensió també es va haver de crear un fluent amb el nombre de pàgines de cada llibre i el nombre de pàgines que s'han llegit un mes per tal de que en la precondició d'aquest operador no es llegeixi un llibre si al llegir-lo es superessin les 800 pàgines que hi ha com a límit superior en un mes. L'efecte d'aquest operador és simplement marcar el llibre com a llegit (`(read ?b)`, guardar en el predicat `(assigned ?b ?m)` en quin mes s'ha llegit i per últim augmentar les pàgines llegides en aquell mes tenint en compte les pàgines del llibre que volem llegir.

Una vegada teníem aquest codi, ja era suficient per tal que funcionés de forma correcta, però ràpidament vam veure que fàcilment es podria optimitzar si afegíssim un parell de precondicions més per tal de poder retallar l'espai de cerca. Aquestes precondicions van ser que el llibre no s'ha-

gués llegit ja en el passat i que, o havia de ser objectiu o predecesssor d'un altre llibre o paral·lel. D'aquesta manera ens assegurem que si un llibre està al catàleg però no el volem llegir ni està relacionat amb algun llibre que sí que sigui el nostre objectiu, no el tindrem en compte. Amb aquesta millores el nostre codi ràpidament va tornar-se molt més eficient ja que anteriorment podríem tenir un catàleg de 100 llibres però només voler llegir-ne un parell i, probablement hauria de passar per molts dels llibres per arribar a l'objectiu o inclús en el planficador resultant podria fer-nos llegir un llibre que no volíem ni fes falta llegir segons les condicions de l'enunciat.

Finalment ens faltava un operador per poder gestionar els salts cronològics en el temps, aquest és `(start_month)`. No disposa de cap precondició a banda del mes actual, anterior i futur. El seu efecte és simplement marcar el més actual com a anterior, el més futur com a actual i l'antic mes anterior treure'l de mes anterior ja que hem avançat un mes. Com que no tenim cap condició per tal de saltar de mes és el propi planificador que s'encarregarà de decidir quan cal saltar i quan no. Com que canviar de mes no l'apropa directament a l'objectiu final que és llegir-se tots els llibres objectius, només en fa ús en el moment en que en un mes no es pot llegir més llibres degut a les limitacions de les pàgines i, per tant, necessàriament cal canviar de mes o degut a que els llibres predecessors s'han de llegir en mesos diferents.

Una vegada ja havíem complert totes les extensions, vam decidir anar un pes més endavant i intentar fer ús dels optimitzadors. Per fer-ho vam desenvolupar un nou fluent el qual cada cop que es canviés de mes s'augmentaria en 1. El nostre objectiu era intentar minimitzar el nombre de mesos del planificador (veure l'experiment 4.1.3).

Així doncs, un cop vam implementar els codis per a la última extensió, vam decidir treure les següents coses per a cada extensió anterior:

- Extensió 2: Eliminem totes les variables (*pages_read, total_pages, num_months_created*). Per tant, ja no minimitzem cap mètrica i ens estalviem tota la part d'inicialitzar les variables en l'arxiu del problema, així com el fet d'incrementar-les dins de les accions.
- Extensió 1: Eliminem el predicat `(parallel ?b1 - book ?b2 - book)`, ja que els llibres no poden tenir paral·lels. De la mateixa manera, eliminem les sentències que feien referència a aquest predicat.
- Nivell Bàsic: Com que cada llibre només pot tenir 0 o 1 llibre predecessor, en les precondicions de l'acció *assign_book* només cal comprovar

3.4 Jocs de prova

Per comprovar si el domini PDDL és correcte, cal crear jocs de prova, executar-los i comprovar que el resultat sigui coherent i el temps d'execució no sigui excessiu. En el nostre projecte, incloem dos jocs de prova generats manualment per cada nivell de complexitat (bàsic, extensió 1, extensió 2 i extensió 3) per poder fer algunes execucions sense necessitat de crear nous jocs de prova. A més, també incloem un generador de jocs de prova aleatoris (subjecte a paràmetres o restriccions que l'usuari pot modificar) per generar i executar ràpidament diferents jocs de prova.

3.4.1 Jocs de prova creats manualment

En la entrega d'aquest projecte s'hi inclouen jocs de prova realitzats "manualment" per tal de comprovar si el nostre planner funciona correctament (és a dir, per veure si hem definit el domini

PDDL correctament). Concretament, s'inclouen dos jocs de prova (arxius problem de PDDL) per cada nivell d'extensió (bàsic, extensió 1, extensió 2 i extensió 3). En cada un d'aquests jocs de prova s'intenta posar a prova les característiques que permet el seu nivell d'extensió en particular, per així poder comprovar que no hi ha errors. Aquests jocs de proven s'anomenen `default_problem_ext_X-v1.pddl` i `default_problem_ext_X-v2.pddl`, sent X él numéro de l'extensió (0, 1, 2 o 3), i cadascun d'aquests arxius de problemes tenen també el seu corresponent arxiu de resultats (output), els quals són uns documents de text anomenats `default_results_ext_X-v1.txt` i `default_results_ext_X-v2.txt`.

<u>Nivell bàsic</u>

Versió 1: L'arxiu de problema de la versió 1 del nivell bàsic intenta provar que funciona correctament el fet que, per haver de llegir un `goal_book`, l'usuari ha d'haver llegit el seu predecessor en un mes anterior a l'actual, i que si aquest predecessor té un altre predecessor que tampoc ha llegit, també l'ha de llegir en un mes anterior, i així succesivament. Així doncs, hem optat per crear aquest arxiu on es definexen 8 llibres, en que el llibre 1 és predecessor del 2, el 2 és predecessor del 3, el 3 ho és del 4, i així fins el vuitè llibre. Hem declarat que l'únic llibre que l'usuari vol llegir es és llibre 6 i que fins al moment no ha llegit cap llibre del catàleg.

Com que el llibre 6 té un predecessor que l'usuari encara no ha llegit, que és el llibre 5, aquest haurà de ser assignat en un mes anterior al del 5, i així successivament pels sis primers llibres, ja que formen una cadena de predecessors que l'usuari necessita llegir per tal de poder llegir el llibre 6, que és el que en un inici volia llegir. Per tant, el resultat hauria de ser que s'han assignat els sis primers llibres (en ordre), i cadascun en un mes diferent, ja que un llibre no pot ser assignat al mateix mes que el seu predecessor.

Si mirem l'arxiu dels resultats, ens fixem que funciona correctament, ja que els llibres estan assignats en l'ordre correcte i cadascun en un mes diferent, per la qual cosa podem veure que la condició dels llibres predecessors per a aquesta extensió funciona correctament.

Versió 2: Per altra banda, en la versió 2 volem provar que, en crear dues cadenes de llibres predecessors que siguin independents entre elles, s'assignin també de manera correcta i es continuï respectant la condició de llegir un predecessor en un mes anterior al del llibre que l'usuari vol llegir. A més, en aquesta versió hem afegit l'escenari en el qual l'usuari ja ha llegit en un passat alguns llibres del catàleg. En concret hem creat 10 llibres, en què hi ha una cadena de predecessors del 1 al 5, i una altra del 6 al 10. L'usuari, però, ja ha llegit anteriorment els llibres 2 i 6, i els llibres que vol llegir són el 5 i el 10. Com podem observar, si funciona correctament, l'usuari hauria de llegir, per a la primera cadena de predecessors, els llibres 3, 4 i 5, en aquest ordre, ja que el llibre predecessor del 3, que és el 2, ja estat llegit per l'usuari i per tant pot assignar el 3 al primer mes. Per la segona cadena de predecessors, s'haurien d'assignar tots els llibres menys el primer de la cadena, ja que és el que l'usuari ja ha llegit en un passat, i es tracta del llibre 6. Per tant, s'haurien d'assignar els llibres en l'ordre mencionat, i no pot haver cap mes en que s'hagin assignat 2 llibres en què un sigui predecessor de l'altre.

Si mirem el resultat, podem veure com, efectivament, s'ha complert el que havíem predit i, per tant, podem confirmar que funciona correctament el nivell bàsic.

A més, podem observar en els dos fitxers de resultats que els temps d'execució obtinguts han estat

0.01 segons tots dos, la qual cosa ens indica que s'arriba a la solució de manera molt ràpida.

Extensió 1

Com que aquesta extensió afegeix l'opció de poder tenir fins a N predecessors, volem comprovar que funciona de forma correcta el fet que un `goal_book` tingui N predecessors, i aquests s'assignin en un mes anterior al llibre objectiu. Per a comprovar-ho, hem augmentant les dues versions del nivell bàsic fins per tal de tenir 15 llibres.

Versió 1: En aquesta primera versió, hem fet que els 15 llibres formin una cadena de predecessors, excepte els llibres 5, 6, 7, 8, 9 i 10, que són tots predecessors del llibre 11. L'únic `goal_book` és el llibre 12 i, per tant, s'hauran d'assignar tots els llibres anteriors a aquest en mesos anteriors. El canvi que hem de veure en aquesta nova versió, és que els llibres que són predecessors del llibre 11 s'assignin al mateix mes, ja que com que no són predecessors entre ells, no tenen per què assignar-se en mesos diferents. Per tant, si veiem el fitxer de resultats podem observar clarament com ha funcionat de manera correcta, ja que tots els llibres s'assignen en l'ordre correcte, i els llibres 5, 6, 7, 8, 9 i 10 (tots predecessors del 11) s'assignen al mateix mes.

Versió 2: En aquesta versió hem definit que els primers 10 llibres són una cadena de predecessors, i en concret els 5 primers són tots predecessors del llibre 6. Per altra banda, els llibres 11, 12 i 13 són predecessors del llibre 14, que a la vegada és predecessor del llibres 15. Els llibres que l'usuari vol llegir són el 6 i el 15, i ja ha llegit en un passat el llibre 2, per la qual cosa s'haurien d'assignar els llibres 1, 3, 4 i 5 al mes anterior al llibre 6, i els llibres predecessors al 14 s'haurien d'assignar al mes anterior a aquest, que a la mateixa vegada també s'hauria d'assignar abans que el mes 15. Si veiem els resultats s'assignen al gener els llibres 11, 12 i 13, mentre que el 14 s'assigna a febrer i el 15 al març, per tant aquesta part funciona correctament. Quant a l'altra cadena de predecessors, els llibres 1, 2, 3, 4 i 5, que són predecessors al 6, s'assignen també al març i aquest últim a l'abril. Per tant, podem confirmar que amb N predecessors també es compleixen les condicions de llibres predecessors.

Les dues versions han tingut uns temps d'execució de 0.03 i 0.02 segons, respectivament, per la qual cosa el planificador segueix trobant la solució en molt poc temps.

Extensió 2
L'extensió 2 afegeix l'opció de que cada llibre pot tenir de 0 a M llibres parallels, per la qual cosa necessitem comprovar que funcionin correctament les condicions especificades a l'enunciat, que són que per a tots els llibres del pla de lectura es satisfà en tot moment que els seus parallels es llegeixen el mateix mes o en el mes anterior. És important remarcar que en el cas que es necessiti llegir un cert llibre, digem-li `book4`, malgrat no sigui `goal_book`, és a dir, sigui predecessor d'un `goal_book` directa o indirectament, si aquest té un llibre parallel sempre haurem de definir el predicat de la manera següent: `(parallel book4 book20)`, en què el llibre que necessitem llegir sempre serà el primer dels dos a l'hora de definir el predicat, ja que el nostre codi està pensat de manera que només es llegirà `book5` si ho definim d'aquesta manera.

Versió 1: Així doncs, agafant la versió 1 de l'extensió anterior, en què tenim 15 llibres, l'hem modificat per tal que el llibre 6 sigui paral·lel als llibres 7, 8, 9 i 10, i el llibre 1 i 2 siguin paral·lels entre ells. El llibre que l'usuari vol llegir és el 6 i, per tant, el planificador hauria d'assignar els llibres paral·lels durant el mateix mes o anterior al llibre 6, i prèviament s'haurien d'haver assignats

els llibres predecessors, és a dir, el 2, 3, 4 i 5, seguint aquest ordre (a més, també s'hauria d'assignar el llibre 1 en el mateix mes o en un mes consecutiu al llibre 2 ja que són paral·lels). Observant el fitxer de sortida, podem veure com primer s'assignen els llibres predecessors al `goal_book` (llibre 6), on també s'ha assignat el llibre 1 al mateix mes que el llibre 2, i tots els llibres que són paral·lels al llibre 6 s'han assignat al mes anterior a aquest, per la qual cosa es compleix en tot moment les condicions sobre llibres paral·lels i predecessors, confirmant que el codi funciona correctament.

Versió 2: En aquesta segona versió volem provar que segueixi funcionant correctament en un escenari més complex, en què ara hi ha dos grups de llibres paral·lels, un format pels llibres 1, 2, 3, 4, 5 i 6, i l'altre format pel 7, 8, 9, i 10. A la mateixa vegada, el 10 també pertany a una cadena de predecessors, que va des del propi llibre 10 fins al 15, i els llibres `goal_book` són l'1 i el 15. Així doncs, mirant els resultats, veiem que al mes de gener s'assignen els primers 10 llibres, que es tracten dels dos grups diferents de parallels. A partir de febrer s'assignen també correctament i seguin l'ordre correcte els llibres que formen la cadena de predecessors, arribant a llegir el llibre 15 al mes de juny, complint així amb el *goal* de llegir els dos llibres `goal_book`.

Observem també que els temps d'execució obtinguts són 0.56 i 0.29 segons, havent augmentat considerablement respecte l'extensió anterior, però segueix sent un càlcul prou ràpid per trobar la solució.

Extensió 3
Aquesta extensió és la més complexa, ja que té en compte el nombre de pàgines de cada llibre per tal de no excedir el límit de 800 pàgines llegides per l'usuari mensualment. Per tal de veure que els fluents que hem creat funcionen correctament, provarem que els resultats obtinguts en les següents dues versions són els esperats, i a més es segueixen complint totes les condicions i restriccions d'apartats anteriors.

Versió 1: Aquesta primera versió de la extensió 3 solament busca provar que els llibres s'assignen de manera correcta sense excedir el límit de pàgines llegides per mes, és a dir, crearem diversos llibres independents (sense relacions de predecessors o parallels) i veurem al assignar-los es satisfà la restricció establerta. Per tal de comprovar-ho, agafant el model de la versió anterior, hem eliminat tots els predicats `parallel` i `predecessor`, i hem afegit la sentència `(= (total_pages book1) 100)` per definir quantes pàgines té cada llibre, en què hem constatat que els primers 8 llibres tenen 100 pàgines, els llibres 9, 10, 11 i 12 tenen 200, els llibres 13 i 14 400, i finalment el llibre 15 té 500. Ara hem de comprovar que no hi hagi cap mes en el qual se li hagin assignat llibres de manera que sumant les pàgines s'excedeixi el límit de 800.

Un cop executat el fitxer, veiem que al gener s'han assignat els llibres 12, 8 i 15, que sumen un total de 200 + 100 + 500 = 800 pàgines. Al febrer s'han assignat els llibres 14 i 13 (400 + 400 = 800), al març els llibres 6, 7, 9, 10, 11 (100 + 100 + 200 + 200 + 200 = 800) i finalment a l'abril s'han assignat els llibres 1, 2, 3, 4 i 5 (que tots tenen 100 pàgines: 100*5 = 500 ¡ 800). Com podem observar, en cap mes s'excedeix el límit imposat i, per tant, podem afirmar que es compleix la restricció de les pàgines llegides mensualment, almenys pels casos en què no hi ha llibres paral·lels ni predecessors, ara falta comprovar si es compleix en casos més complexos.

Versió 2: En aquesta segona versió volem veure si, en un problema com el de la versió 2 de l'extensió 3 (en què hi ha dos grups de llibres parallels i una cadena de predecessors), no s'incompleix la restricció de 800 pàgines mensuals, i veure com això afecta a la creació de mesos. Per tant, a la versió anterior hem inicialitzat els valors de les pàgines de cada llibre de manera que el 15 llibres

tenen, respectivament, les següents pàgines: 250, 200, 150, 300, 250, 400, 550, 200, 150, 450, 600, 500, 250, 300 i 450. Cal tenir en compte que si posem un número molt gran de pàgines per a molts llibres, pot ser que el planificador no pugui trobar una solució ja que, o bé s'incomplirien les regles dels llibre predecessors i parallels, o bé es necessitarien més mesos per tal de fer el pla de lectura, per la qual cosa no obtindríem solució. Amb aquest exemple, un cop l'executem, obtenim l'assignació de llibres següent per a cada mes diferent:

- Gener: `Book_7` i `Book_9` (550 + 150 = 700).
- Febrer: `Book_8` i `Book_10` (200 + 450 = 650).
- Març: `Book_3`, `Book_5` i `Book_6` (150 + 250 + 400 = 800).
- Abril: `Book_1`, `Book_2` i `Book_4` (250 + 200 + 300 = 750).
- Maig: `Book_11` (600).
- Juny: `Book_12` (500).
- Juliol: `Book_13` (250).
- Agost: `Book_14` (300).
- Setembre: `Book_15` (450.

Veiem que el grup de paral·lels format pels llibres 1, 2, 3, 4, 5 i 6, estan tots assignats en dos mesos consecutius (març i abril), i l'altre grup de paral·lels (llibres 7, 8, 9 i 10) també estan assignats en dos mesos seguits (gener i febrer), per la qual cosa es compleix la condició de llibres paral·lels. Per altra banda, podem veure que la cadena de predecessors també satisfà les condicions, ja que cada predecessor està assignat a un mes anterior al seu successor. Finalment, en cap mes s'incompleix el fet de llegir més de 800 pàgines, per lo tant, podem afirmar que el nostre codi funciona correctament, ja que els resultats per a totes les extensions compleixen amb les condicions especificades a l'inici de la pràctica.

Per últim, si ens fixem amb els temps obtinguts en aquesta última extensió, podem veure clarament que en augmentar la complexitat del problema, el temps d'execució es veu incrementat significativament, ja que hem obtingut, per cada versió, 0.03 i 1.98 segons respectivament, lo qual ens indica que a mesura que anem afegint més predicats de `parallel` i `predeccessor`, el programa trigarà més en executar-se, ja que a causa de l'escalabilitat el temps d'execució creix exponencialment a mesura que el programa és més complex.

3.4.2 Generador de jocs de prova aleatoris en Python

Tal i com es demanava en l'enunciat de la pràctica, també hem creat un generador aleatori de jocs de prova per poder fer execucions de manera molt més ràpida, eficient i pràctica. Aquest generador ha estat creat en Python i es pot trobar a l'entrega en l'arxiu anomenat `generar_jocs_proves.py`.

3.4.2.1 Contingut general i requeriments

El document Python conté aproximadament 650 línies de codi. Inicialment es fan les importacions necessàries i es declaren funcions i classes auxiliars que s'utilitzaran al llarg del programa. Seguidament es fan diverses preguntes a l'usuari per quina mena de jocs de prova es volen generar. A continuació es genera el joc de proves (l'arxiu amb el problem de PDDL) i es pregunta a l'usuari si vol realitzar automàticament l'execució del joc de proves.

Per poder executar el programa Python és necessari tenir instal·lades les llibreries següents:

- `networkx`.
- `numpy`. L'utilitzem principalment per la generació de nombres aleatoris. Preferim utilitzar-lo en comptes de la tradicional llibreria `random` perquè el considerem més còmode a l'hora de treballar amb distribucions estadístiques, però aquesta decisió és simplement una qüestió de preferència.
- `matplotlib`.
- Altres mòduls com `os`, `subprocess` o `platform`, que en teoria venen inclosos per defecte a Python.

3.4.2.2 Estructures de dades utilitzades

Per generar jocs de prova correctes és molt important estructurar les dades de manera clara i amb restriccions que evitin la generació de jocs de prova irresolubles.

En la nostra implementació hem implementat una estructura de grafs dirigits a través de la popular llibreria `NetworkX`. Cada graf dirigit representa els llibres i les relacions entre ells (*parallel* o *predecessor*) per a un joc de proves en particular.

En aquest graf, els nodes són els llibres, representats com a instàncies d'una classe anomenada *Book* i amb atributs `name` (nom del llibre) i `pages` (nombre de pàgines del llibre). Per altra banda, les arestes són les relacions de predecessor o paral·lel entre llibres.

La relació de *predecessor* entre un llibre i un altre està representada mitjançant una aresta que surt del llibre que s'hauria de llegir primer i apunta al llibre que s'hauria de llegir posteriorment. És a dir, si el llibre u s'ha de llegir abans que el llibre v, en el graf hi haurà una aresta que va des d'u fins a v. Aquestes arestes s'afegeixen amb l'atribut[1] `name='predecessor'`.

Per altra banda, per indicar que un llibre és paral·lel a un altre, s'utilitzen també arestes, però aquesta vegada amb l'atribut `name='parallel'`. En aquest cas no caldria utilitzar arestes dirigides (ja que si u és paral·lel a v, llavors v també és paral·lel a u), però això no és possible degut a que el graf és dirigit (objecte `DiGraph()` de `NetworkX`). Es podria representar una aresta no dirigida afegint dues arestes ($u \to v$ i $v \to u$), però això generaria cicles en el graf i més endavant veurem que això no és desitjable. Així doncs, encara que les arestes siguin dirigides, treballarem amb elles com si no ho fossin, sense importar el sentit de l'aresta.

[1] En `NetworkX` les arestes poden tenir atributs, com ara un nom.

3.4.2.3 Funcions utilitzades

A l'inici de l'arxiu Python es declaren 3 funcions molt importants per garantir que es creen jocs de prova correctes (resolubles).

- `correct_compilation_on_mac()`: aquesta funció retorna `True` si troba l'arxiu 'ff' (arxiu executable Unix) en la ruta `./executables/macos`. En cas contrari retorna `False`. Aquest arxiu indica que en MacOS s'ha compilat correctament el programa 'ff' i per tant es pot realitzar l'execució del programa. En cas contrari, abans d'executar el joc de proves realitzarà la comanda `make ff` per compilar el programa.

- `has_cycle(graph, u, v, cycle_type)`: aquesta funció retona `True` si l'addició de l'aresta `u` $\rightarrow$ `v` en el graf dirigit `graph` genera un cicle (per comprovar si hi ha un cicle, `cycle_type='directed'` té en compte el sentit de les arestes, mentre que `cycle_type='undirected'` considera les arestes com no dirigides). En cas contrari retorna `False`.

  Aquesta funció no es crida directament des del programa, sinó que és una funció auxiliar de `add_edge_if_no_cycle()`.

- `add_edge_if_no_cycle(graph, u, v, edge_name, cycle_type)`: aquesta funció afegeix l'aresta `u` $\rightarrow$ `v` al graf dirigit `graph` en cas que la seva addició no generi un cicle (per comprovar si hi ha un cicle, `cycle_type='directed'` té en compte el sentit de les arestes, mentre que `cycle_type='undirected'` considera les arestes com no dirigides). L'atribut de l'aresta afegida serà `name='predecessor'` (quan `edge_name='predecessor'`) o `name='parallel'` (quan `edge_name='parallel'`).

  Retorna `True` si ha pogut afegir l'aresta, `False` en cas contrari (informació que posteriorment s'utilitza per tenir un recompte de les arestes afegides i deixar d'afegir-ne quan s'ha arribat al límit). la funció s'utilitza cada vegada que es vol afegir una aresta a graf per garantir que no s'afegeixen arestes que provoquen cicles. Això ens permet crear jocs de proves on, per exemple: si u és predecessor de v, u i v no siguin paral·lels de cap manera, ni v pugui ser predecessor d'u, ja que qualsevol d'aquests casos seria incoherent amb les pròpies definicions de predecessor i de paral·lel.

- `parallel_chained_nodes(graph, initial_node)`: aquesta funció retorna un conjunt (*set* en Python) amb tots els nodes (instàncies de la classe *Book*) del graf dirigit `graph` que formen part de la mateixa "cadena" de llibres paral·lels que `initial_node`, incloent `initial_node`. Considerem que un node u pertany a la mateixa "cadena" de llibres paral·lels que un node v si i només si existeix un camí d'arestes amb atribut `name='parallel'` que uneix u amb v (les arestes es consideren no direccionals; és a dir, el camí pot recórrer una aresta en sentit contrari).

  Aquesta funció s'utilitza per verificar restriccions dins del mateix grup de llibres paral·lels (com ara que sigui possible llegir-los en 2 mesos, que no s'afegeixi l'aresta en cas de ja estar en el mateix grup de paral·lels, ...).

- `is_semi_goal(graph, node, goal_books)`: aquesta funció retorna `True` en cas que `node` sigui un node "`semi_goal_book`" en el graf dirigit `graph`. Un node "`semi_goal_book`" és aquell que té algun predecessor (mitjançant només arestes de tipus *predecessor*) que és `goal_book`. En cas contrari, retorna False. Aquesta funció és útil, com s'explica més endavant, per determinar el node "arrel" d'un mateix grup (o "cadena") de nodes paral·lels.

3.4.2.4 Paràmetres i interacció amb l'usuari

El programa de generació de jocs de prova s'ha fet des d'un principi pensat perquè l'usuari pugui controlar còmodament com es generen els jocs de prova. Per això, en l'execució del programa es fan diverses preguntes a l'usuari que han de ser respostes mitjançant el propi terminal. A més, algunes d'aquestes preguntes es poden respondre escrivint *HELP* per obtenir una explicació més detallada de la pregunta perquè l'usuari sàpiga exactament què pot respondre i com la seva resposta afectarà a la generació de jocs de prova. Cal mencionar que les preguntes que es fan varien en funció de les respostes anteriors de l'usuari i que el codi és totalment robust a respostes no vàlides per part de l'usuari (en cas de resposta no vàlida es mostra un missatge amb les condicions que ha de complir la seva resposta i es torna a repetir la pregunta, però en cap cas s'acaba l'execució del programa degut a un error en la resposta). A continuació s'expliquen les preguntes que es realitzen a l'usuari i com afecten les respostes a la generació del joc de proves.

1. **Nivell d'extensió dels jocs de prova**: es pregunta a l'usuari quin és el nivell d'extensió dels jocs de prova que vol generar (B: nivell bàsic, 1: extensió 1, 2: extensió 2, 3: extensió 3).

2. **Nombre de jocs de prova**: es pregunta a l'usuari quants jocs de prova del nivell d'extensió triat vol generar.

3. **Sagues de llibres conegudes**: es pregunta a l'usuari si vol incloure sagues de llibres conegudes en els seus jocs de proves. Aquestes sagues són conjunts de llibres amb les relacions de predecessors i paral·lels ja establertes segons són a la realitat (menys la saga de *Marvel*, que ha estat inventada per nosaltres). Les sagues de llibres disponibles són les següents:

   - **Harry Potter**: aquesta saga es pot afegir al joc de proves incloent *HP* a la resposta. Conté relacions de llibres predecessors. Es pot visualitzar en la Figura [1].
     - *HP1_Harry_Potter_and_the_Philosophers_Stone* (220 pàgines).
     - *HP2_Harry_Potter_and_the_Chamber_of_Secrets* (250 pàgines).
     - *HP3_Harry_Potter_and_the_Prisoner_of_Azkaban* (315 pàgines).
     - *HP4_Harry_Potter_and_the_Goblet_of_Fire* (635 pàgines).
     - *HP5_Harry_Potter_and_the_Order_of_the_Phoenix* (800 pàgines).
     - *HP6_Harry_Potter_and_the_Half-Blood_Prince* (600 pàgines).
     - *HP7_Harry_Potter_and_the_Deathly_Hallows* (600 pàgines).
   - **The Lord of the Rings (El Senyor dels Anells)**: aquesta saga es pot afegir al joc de proves incloent *LR* a la resposta. Conté relacions de llibres predecessors. Es pot visualitzar en la Figura [2].
     - *LR1_The_Fellowship_of_the_Ring* (420 pàgines).
     - *LR2_The_Two_Towers* (350 pàgines).
     - *LR3_The_Return_of_the_Kin*g (415 pàgines).
   - **The Hunger Games (Els Jocs de la Fam)**: aquesta saga es pot afegir al joc de proves incloent *HG* a la resposta. Conté relacions de llibres predecessors. Es pot visualitzar en la Figura [3].
     - *HG1_The_Hunger_Games* (370 pàgines).
     - *HG2_Catching_Fire* (390 pàgines).
     - *HG3_Mockingjay* (390 pàgines).

- **Marvel**: aquesta saga es pot afegir al joc de proves incloent MV a la resposta. Només està disponible en els nivells d'extensió 2 i 3, ja que conté relacions de llibres predecessors i també de paral·lels. Es pot visualitzar en la Figura [4].
  - *Avengers_1* (200 pàgines).
  - *Avengers_2* (250 pàgines).
  - *Avengers_3* (500 pàgines).
  - *Ironman* (300 pàgines).
  - *Hulk* (250 pàgines).
  - *Captain_America* (350 pàgines).
  - *Thor* (400 pàgines).
  - *Black_Widow* (300 pàgines).
  - *Hawkeye* (250 pàgines).
  - *Spiderman_1* (250 pàgines).
  - *Spiderman_2* (300 pàgines).
  - *Guardians_of_the_Galaxy_*1 (300 pàgines).
  - *Guardians_of_the_Galaxy_2* (550 pàgines).
  - *Guardians_of_the_Galaxy_3* (250 pàgines).
  - *Groot* (400 pàgines).

4. **Llibres addicionals**: es pregunta a l'usuari quants llibres addicionals vol generar. Aquests llibres no tindran relacions amb cap de les sagues de llibres famoses que l'usuari hagi decidit afegir. Les relacions de predecessor (i també paral·lel en el nivell d'extensions 2 i 3) es generaran aleatòriament tenint en compte les restriccions que no permeten generar jocs de proves incorrectes.

5. **Percentatge de relacions entre llibres**: es pregunta a l'usuari quin percentatge d'arestes vol afegir respecte el total possible ($K-1$, on K és el nombre de llibres addicionals, ja que el graf és acíclic, com es mencionarà més endavant). És a dir, es pregunta per la quantitat de relacions que tindran els llibres en el graf. Si l'usuari introdueix 100, s'afegiran totes les arestes (de moment sense determinar si de tipus *predecessor* o *parallel*) possibles ($K-1$). En canvi, si l'usuari introdueix qualsevol altre valor x, s'afegiran $\frac{x}{100} \times (K-1)$ arestes. Així doncs, podem dir que aquesta pregunta serveix per determinar si el graf estarà més o menys connectat, o si tindrà més o menys components connexos.

6. **Percentatge de relacions *parallel***: en els nivells d'extensió 2 i 3, una vegada s'ha determinat el nombre d'arestes totals que s'afegiran, es pregunta a l'usuari per, dins d'aquesta quantitat escollida prèviament, quin percentatge d'arestes han de ser de tipus *parallel* (s'assumeix que les arestes restants seran de tipus *predecessor*). A l'hora de respondre aquesta pregunta, s'ha de tenir en compte que les arestes *parallel* generalment generen problems més complexos que les *predecessor*, de manera que un valor a respondre raonable normalment està per sota de 20.

7. **Percentatge de llibres a llegir (`goal_book`)**: es pregunta a l'usuari quin percentatge de llibres ha de ser escollit aleatòriament per llegir en el planner. En cas que es vulguin llegir tots els llibres (per eliminar el factor aleatòri, per exemple) cal respondre amb un 100 (el màxim).

8. **Llavor del generador aleatori**: en cas que l'usuari hagi decidit afegir més d'un llibre aleatori o bé hagi decidit que els llibres a llegir es determinin aleatòriament, es pregunta a l'usuari quina llavor (*seed*) vol utilitzar pel generador de nombres aleatoris (`numpy.random`). Si introdueix el valor 0 s'escollirà una llavor qualsevol.

9. **Execució automàtica del planner**: es pregunta a l'usuari si vol executar automàticament els jocs de prova que ha generat. En cas negatiu, s'acaba l'execució. En cas afirmatiu, es procedeix a la següent pregunta per concretar els detalls de l'execució.

10. **Optimitzador de mesos**: en cas que l'usuari vulgui executar automàticament els jocs de prova, se li pregunta si vol incloure el flag `-O` a l'execució. Aquest flag, tal i com es declara en el domini afegint `(:metric minimize (num_months_created))`, farà que es minimitzi el màxim possible el nombre de mesos creats. És a dir, si es pot realitzar un pla de lectura en 5 mesos, no ho farà en més (com passa a vegades si no s'afegeix el flag `-O`). Això és útil en cas de voler el resultat més òptim pel pla de lectura, però augmenta molt el cost computacional i el temps d'execució (especialment en jocs de prova on hi ha molts llibres).

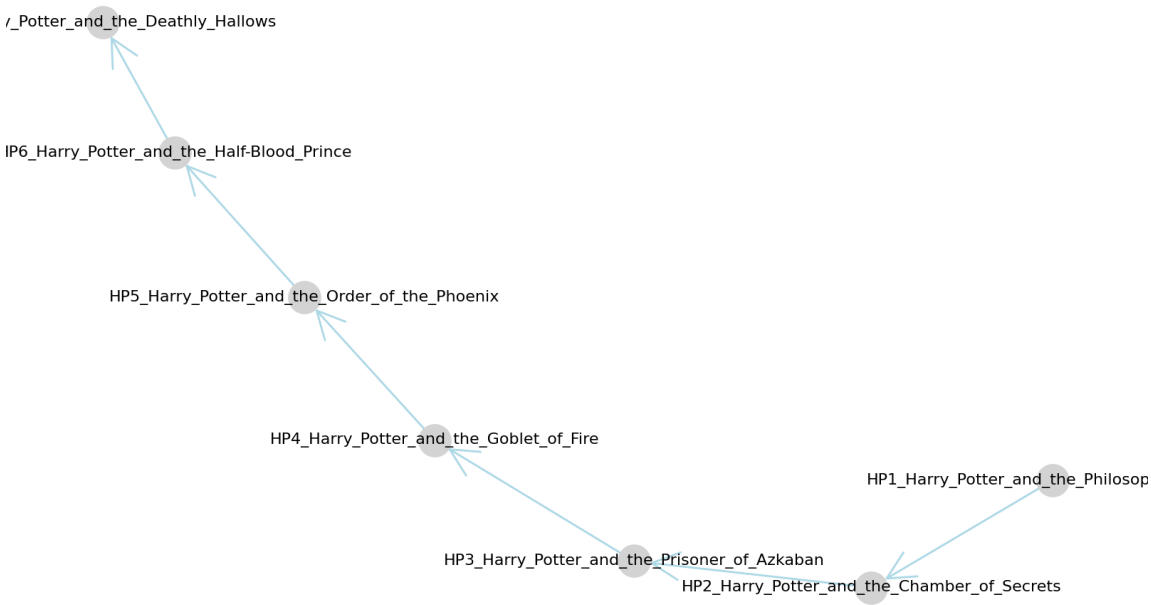

Figura 1: Visualització de la saga de llibres de Harry Potter. Les arestes blaves representen les relacions de predecessor.

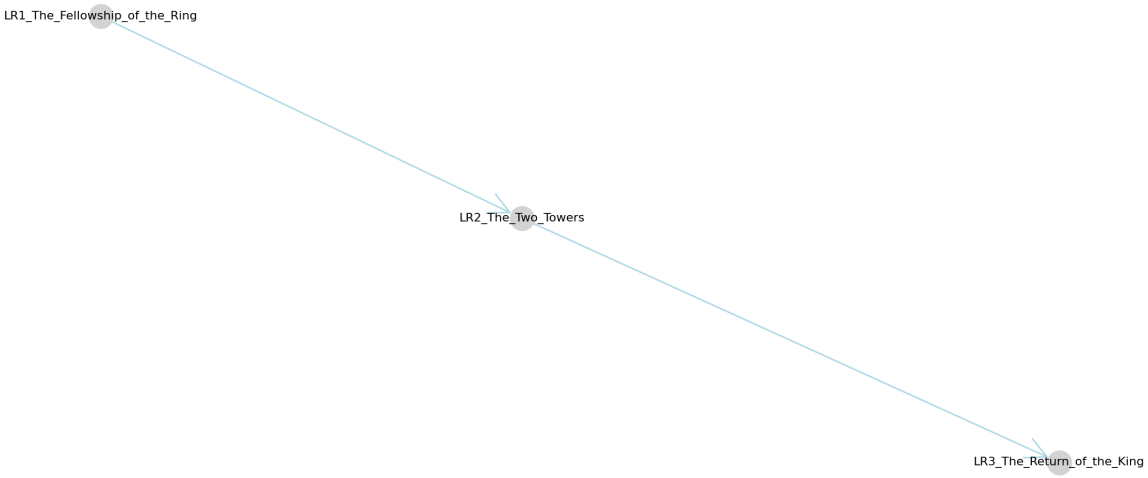

Figura 2: Visualització de la saga de llibres de The Lord of the Rings (El Senyor dels Anells). Les arestes blaves representen les relacions de predecessor.

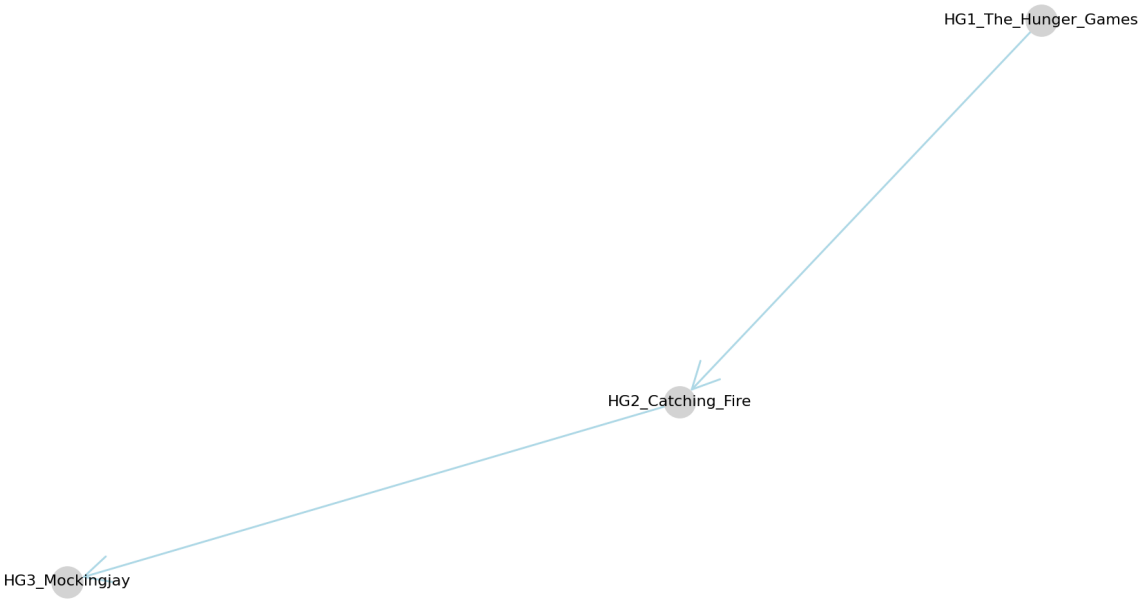

Figura 3: Visualització de la saga de llibres de The Hunger Games (Els Jocs de la Fam). Les arestes blaves representen les relacions de predecessor.

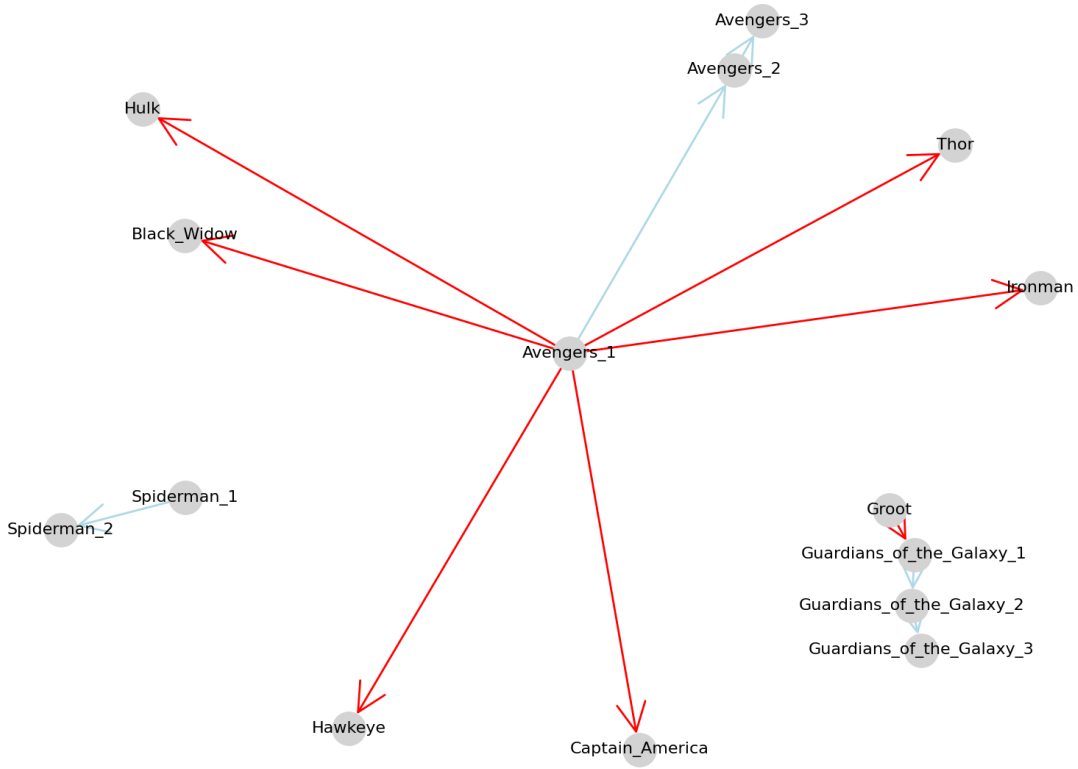

Figura 4: Visualització de la saga de llibres de Marvel. Les arestes blaves representen les relacions entre llibres predecessors, mentre que les vermelles representen les relacions entre llibres paral·lels. En les relacions de paral·lel (arestes vermelles) el sentit de les arestes no és rellevant ($u \to v$ és el mateix que $v \to u$ i simplement indica que u i v són paral·lels.)

3.4.2.5 Restriccions

Com ja hem anat mencionant, en el nostre generador de jocs de prova només es generen jocs de prova resolubles (excepte si l'usuari introdueix paràmetres extrems, com ara introduir una quantitat excessiva de llibres addicionals al joc de proves de manera que no sigui possible llegir-los en els 12 mesos, etc.). A continuació s'expliquen algunes de les restriccions que el programa considera per generar bons jocs de proves.

- **Grafs acíclics**:

  Els grafs dirigits que s'utilitzen per representar les relacions de cada joc de proves no poden ser grafs amb cicles de cap tipus. És a dir, cada vegada que s'intenta afegir una aresta es comprova que aquesta aresta no generi cap tipus de cicle. Tot això es fa mitjançant la funció `add_edge_if_no_cycle()` donant sempre el paràmetre `cycle_type='undirected'` (perquè no consideri el sentit de les arestes, i així sigui més restrictiu; és a dir, no permeti cicles de cap tipus, encara que el graf sigui dirigit).

Amb aquesta restricció s'evita que un llibre u sigui predecessor i a la vegada paral·lel a un llibre v, o que hi hagi dos o més camins a dins del mateix grup de paral·lels indicant que un llibre i és paral·lel a un altre llibre j (amb un sol camí és suficient), o que un llibre x sigui predecessor a un llibre y i a la vegada y sigui predecessor a x, etc.. En tots aquests casos, quan ens referim a predecessors o a paral·lels, no fem referència a una relació directa (una aresta que connecta directament un node amb un altre), sinó que ens referim a que hi hagi un camí d'arestes *predecessor* o *parallel*, respectivament, que porti d'un node a l'altre.

- **Grups de paral·lels amb nombre de pàgines limitat**:

  Segons les condicions de l'enunciat de la pràctica: "*[...] para todos los libros del plan se cumple en todo momento que sus paralelos se leen en el mismo mes o en el mes anterior (o en el mes siguiente, para la relación simétrica) [...]*". És a dir, tots els llibres que pertanyin a un mateix grup (o "cadena") de llibres paral·lels[2] (anomenat P d'ara en endavant) s'han de llegir en dos mesos consecutius. Així doncs, quan afegim arestes aleatòries de tipus *parallel* entre llibres en el nivell d'extensió 3 (l'únic on es considera el nombre de pàgines dels llibres) hem de procurar que tots els llibres de P puguin ser ser assignats d'alguna manera en dos mesos consecutius. Per garantir aquesta condició, vam plantejar diferents solucions que es mencionen a continuació.

  Inicialment, vam considerar la restricció de que $\forall u, v \in G$ només s'afegís l'aresta $u \to v$ si, una vegada afegida, la suma de les pàgines de P (grup de llibres paral·lels a u/v, incloent u/v) fos igual o inferior a 1600 pàgines (2 mesos $\times$ 800 pàgines). No obstant, pot haver casos on la suma de pàgines sigui igual o inferior a 1600 però no existeixi cap distribució dels llibres de P dos conjunts, C_1 i C_2, on la suma de pàgines de cada conjunt C_i sigui igual o inferior a 800. La condició de l'enunciat només es podria garantir quan la suma de pàgines dels llibres de P fos igual o inferior a 800, ja que llavors hi cabrien en un sol mes i no seria necessari distribuir-los en dos mesos. No obstant, no ens sembla una bona opció limitar a 800 la suma de pàgines de P, ja que es simplifica excessivament la resolució del problema i es restringeix la generació de certs jocs de prova que haurien de ser vàlids.

  Una altra opció és comprovar si realment existeix una combinació (distribució) de llibres de P en els conjunts C_1 i C_2 on la suma de pàgines dels llibres de cada conjunt C_i sigui igual o inferior a 800. No obstant, aquest càlcul correspon a una variant (o cas específic) del conegut Problema de la Motxilla (*Knapsack Problem*) [1], bastant similar al problema d'Empaquetament Binari (*Bin Packing*) [2]. Aquests problemes són NP-Complet [3] i NP-Difícil [4], respectivament, de manera que van acompanyats d'un gran cost computacional i temporal. En el nostre cas, hauríem de comprovar $\mathrm{O}(2^k)$ combinacions diferents, on k és el nombre de llibres del conjunt P. Degut a la complexitat d'aquesta tasca, en el nostre programa hem decidit deixar la condició de que la suma de pàgines de tots els llibres de P sigui igual o inferior a 1600. Així doncs, hem de mencionar que és possible (tot i que poc probable) que algun dels jocs de prova que es generin no siguin resolubles, especialment en casos on la suma de pàgines de tots els llibres de P sigui molt propera a 1600 o hi hagi llibres amb moltes pàgines.

3.4.2.6 Altres consideracions i característiques del generador

[2] Considerem que un grup (o "cadena") de llibres paral·lels és un conjunt de nodes (llibres) P on $\forall u, v \in P$ existeix un camí d'arestes amb atribut `name='parallel'` (relacions de llibres paral·lels) que porta de u a v, sense considerar el sentit de les arestes (com si fossin bidireccionals). En el nostre programa trobem aquests grups mitjançant la funció `parallel_chained_nodes()`, explicada anteriorment.

En els apartats anteriors ja s'han mencionat les assumpcions de la nostra implementació del generador en Python i com afecten als jocs de prova. A continuació s'expliquen algunes de les característiques, detalls o assumpcions de la implementació que no s'han pogut mencionar prèviament.

- **Visualitzador de jocs de prova**:

  En el generador s'inclou un visualitzador de jocs de prova (amb el que s'han generat les figures 1, 2, 3 i 4). Aquest visualitzador el proporciona la pròpia llibreria `NetworkX` mitjançant `matplotlib` i permet veure les relacions entre els diferents llibres del joc de proves. S'executa automàticament per cada joc de proves que generem i s'hi poden veure els nodes (llibres) de color gris, taronja o verd i amb el seu nom, a més d'arestes de color blau o vermell. Els nodes taronjes representen llibres que han estat llegits en el passat (assignats al mes *Past*), els nodes verds representen llibres que l'usuari es vol llegir (`goal_book`) i els nodes grisos representen la resta de llibres que no compleixen les condicions anteriors. Les arestes blaves representen relacions de llibres predecessors, mentre que les vermelles representen relacions de llibres paral·lels. En les de predecessors, el sentit de l'aresta és rellevant ($u \to v$ indica que u és predecessor de v); però en les relacions de paral·lels es pot ignorar el sentit de les arestes ($u \to v = v \to u$ i indica que u i v són paral·lels).

  Per altra banda, en el terminal s'escriuen missatges rellevants sobre quines arestes s'han afegit, el nombre d'arestes de cada tipus, els arxius creats, etc..

- **Assignació dels llibres objectiu i llibres prèviament llegits (*Past*)**:

  Els llibres que es declaren com a `goal_book` (tant si ho són tots com si es determinen aleatòriament) no poden ser assignats al mes *Past* (és a dir, no poden haver estat llegits prèviament per l'usuari). Això ho fem per evitar que els predicats als que ha d'arribar el planner ja estiguin satisfets i, per tant, els resultats del planner aparentin ser més bons del que haurien sigut si l'estat inicial no fos tant favorable.

- **Nombre de relacions entre llibres (arestes)**:

  Hem de tenir en compte que, en el nostre programa, totes les arestes s'afegeixen mitjançant la funció `add_edge_if_no_cycle()` i amb el paràmetre `cycle_type='undirected'` (tal i com s'ha explicat en l'anterior apartat 3.4.2.5). D'aquesta manera treballem sempre amb grafs acíclics (són dirigits, però les restriccions de cicles s'apliquen considerant els grafs com no dirigits). Al treballar amb grafs acíclics, el nombre màxim d'arestes que es poden afegir al graf és $n - k$, on n és el nombre de nodes del graf (nombre de llibres addicionals, en el nostre cas) i k és el nombre de components connexos del graf (pot ser qualsevol nombre $\leq n$, però en el cas on hi hagi més arestes serà quan tot el graf és connex; és a dir, $k = 1$ i el nombre màxim d'arestes és $n - 1$. En aquest cas, es diu que el graf és un arbre). En la nostra implementació, el valor de k depèn totalment del percentatge d'arestes a afegir respecte el total possible que l'usuari introdueixi.

- **Assignació aleatòria de relacions entre llibres (arestes)**:

  L'assignació de relacions entre llibres (ja sigui de tipus *predecessor* o *parallel*) és aleatòria però subjecte als paràmetres introduïts per l'usuari. A continuació s'explica el procés d'assignació per cada nivell d'extensió. Quan es mencioni "intentar afegir una aresta" ens referirem a que s'afegeix l'aresta al graf si i només si la seva addició no viola cap de les restriccions del graf (com, per exemple, que sigui acíclic).

- **Nivell bàsic**: inicialment, es tria una parella de nodes aleatoris $u, v \in G$ (on G és el graf dirigit corresponent al joc de proves). Si $u \neq v$, l'aresta $u \to v$ no existeix en G i v no té cap predecessor, aleshores s'intenta afegir l'aresta $u \to v$. Es repeteix el mateix procés fins que s'hagin afegit tantes arestes com el límit d'arestes (determinat per l'usuari en la pregunta sobre el percentatge s'arestes a afegir respecte el total possible. Com a molt $k - 1$, on k és el nombre de llibres addicionals).
- **Exensió 1**: inicialment, es tria una parella de nodes aleatoris $u, v \in G$ (on G és el graf dirigit corresponent al joc de proves). Si $u \neq v$ i l'aresta $u \to v$ no existeix en G, aleshores s'intenta afegir l'aresta $u \to v$. Es repeteix el procés successivament fins que s'assoleixi el límit d'arestes establert.
- **Extensió 2**: inicialment, es separa el procés en arestes *predecessor* i arestes *parallel*.
  - Arestes *predecessor*: es tria una parella de nodes aleatoris $u, v \in G$ (on G és el graf dirigit corresponent al joc de proves). Si $u \neq v$ i no existeix cap aresta $u \to v$ en G, aleshores s'intenta afegir l'aresta $u \to v$. Es repeteix el procés successivament fins que s'assoleixi el límit d'arestes de tipus *predecessor* establert (determinat per l'usuari tant en el nombre d'arestes totals a afegir com en el percentatge d'arestes *predecessor*/*parallel* escollits).
  - Arestes *parallel*: es tria una parella de nodes aleatoris $u, v \in G$ (on G és el graf dirigit corresponent al joc de proves). Si $u \neq v$ i no existeix cap aresta $u \to v$ en G, aleshores s'intenta afegir l'aresta $u \to v$. Es repeteix el procés successivament fins que s'assoleixi el límit d'arestes de tipus *parallel* establert (determinat per l'usuari tant en el nombre d'arestes totals a afegir com en el percentatge d'arestes *predecessor*/*parallel* escollits).
- **Extensió 3**: inicialment, es separa el procés en arestes *predecessor* i arestes *parallel*.
  - Arestes *predecessor*: es tria una parella de nodes aleatoris $u, v \in G$ (on G és el graf dirigit corresponent al joc de proves). Si $u \neq v$ i no existeix cap aresta $u \to v$ en G, aleshores s'intenta afegir l'aresta $u \to v$. Es repeteix el procés successivament fins que s'assoleixi el límit d'arestes de tipus *predecessor* establert (determinat per l'usuari tant en el nombre d'arestes totals a afegir com en el percentatge d'arestes *predecessor*/*parallel* escollits).
  - Arestes *parallel*: es tria una parella de nodes aleatoris $u, v \in G$ (on G és el graf dirigit corresponent al joc de proves). Si $u \neq v$, u i v no es troben al mateix grup (o "cadena") de paral·lels (és a dir, no són ja paral·lels), no existeix cap aresta $u \to v$ en G i la suma de pàgines del grup de paral·lels que resultarà després d'afegir l'aresta $u \to v$ és igual o inferior a 1600, aleshores s'intenta afegir l'aresta $u \to v$. Es repeteix el procés successivament fins que s'assoleixi el límit d'arestes de tipus *parallel* establert (determinat per l'usuari tant en el nombre d'arestes totals a afegir com en el percentatge d'arestes *predecessor*/*parallel* escollits).

- **Pàgines dels llibres addicionals**:

  El nombre de pàgines dels llibres addicionals (només utilitzat en el nivell d'extensió 3) es decideix aleatòriament seguint una distribució normal amb mitjana de 275 i desviació estàndard de 100 (valors que han sigut escollits arbitràriament). A més, el nombre de pàgines sempre serà com a mínim 15 i com a màxim 800 (més de 800 pàgines no té sentit, ja que no es podria assignar a cap mes).

- **Llibre "arrel" en cada grup de llibres paral·lels**:

  Tal i com s'ha explicat en l'apartat 3.3 (Desenvolupament dels models), és molt important pel nostre planificador que, per cada grup (o "cadena") de K llibres paral·lels $P_i \in G$ (on G és el graf dirigit que representa el joc de proves), cada llibre k_j es declari en el document com a paral·lel a un únic llibre k_k ($j \neq k$) que ha de ser `goal_book` (o "`semi_goal_book`"). En el nostre programa, el llibre k_k (o llibre "arrel") es tria escollint el primer llibre declarat com a `goal_book` de tots els llibres del conjunt P_i. En cas de no trobar-ne cap, es tria el primer llibre declarat com a `semi_goal_book` (que té un predecessor, mitjançant només arestes de tipus *predecessor*, que és goal_book, com ja s'ha explicat en la funció `is_semi_goal()`). Si tampoc se'n troba cap, es tria un dels nodes aleatòriament.

  Finalment, una vegada escollit el llibre "arrel", en l'arxiu PDDL tots els llibres de P_i tenen una aresta amb atribut `name='parallel'` que que va des d'aquell node (llibre) al llibre que s'ha declarat com "arrel" (k_k).

  Cal mencionar que, en el visualitzador que es mostra quan es generen els jocs de prova, la direcció d'aquestes arestes encara no està determinada, ja que el càlcul del llibre arrel es fa posteriorment (justament abans de fer el `file.write()` de les relacions entre llibres paral·lels a l'arxiu del joc de proves).

- **Execució automàtica del planner**:

  S'ha de tenir en compte que, per l'execució automàtica del planner, el Sistema Operatiu ha de ser *Windows* (amb *Powershell* instal·lat) o *MacOS* (amb *Xcode* instal·lat). El programa detecta el Sistema Operatiu mitjançant la llibreria `platform` de Python i per l'execució automàtica no cal que l'usuari descarregui *metricff* o *ff*, ja que en la mateixa entrega hem inclòs els executables perquè el programa pugui localitzar-los correctament i executar-los.

  L'execució del planner es fa mitjançant la llibreria `subprocess` de Python i els resultats del planner es guarden en arxius anomenats `generated_result_ext_X_vY.txt` (on `X` és el nivell d'extensió, `Y` és el número de joc de prova d'extensió `X` que s'ha generat. En cas que s'executi amb el flag `-O` per optimitzar el nombre de mesos generats, s'afegeix també "`opt`" al nom de l'arxiu) que es poden trobar en el directori `./results`.

4 Experiments i anàlisi de la complexitat

A continuació es presenten els experiments realitzats, les seves metodologies i els seus resultats. A més, es planteja un estudi de la complexitat del domini analitzant el nombre d'estats visitats i la profunditat màxima, entre altres factors.

4.1 Experiments

Hem plantejat diversos experiments per poder provar el rendiment del nostre programa. A continuació s'explica com s'ha realitzat cada un dels experiments i quines conclusions s'han extret. Cal mencionar que totes bases de dades (documents `.csv`) amb les dades de les diferents execucions s'inclouen en l'entrega en la carpeta 'experiments' i es poden generar els *plots* executant l'arxiu Python anomentat `generar_plots_experiments.py`. Aquest arxiu Python utilitza les llibreries `pandas`, `matplotlib` i `seaborn` per llegir les dades i realitzar els gràfics, així que cal tenir-les instal·lades abans d'executar-lo.

4.1.1 Primer experiment

En aquest experiment volem comprovar per cada extensió com varia el temps respecte l'augment de llibres. Inicialment, la nostra hipòtesi és que tindrà un creixement exponencial, ja que el nombre de combinacions que ha de comprovar el planner és exponencial respecte el nombre de llibres.
Per iniciar l'experiment vam començar amb l'extensió bàsica. Com que aquesta extensió requereix poca força computacional, vam decidir que en l'experiment hi hauria un 100% d'arestes predecessors per tal de complicar-ho una mica. També vam decidir que no s'hauria llegit cap llibre i que els llibres objectiu fossin tots.

Inicialment vam començar amb 5 llibres i vam decidir que aniríem augmentant de 5 en 5 fins als 100 per poder veure com creix el temps o, si més no, trobar una certa tendència. Per fer-ho vam fixar la llavor *42* i realitzar 5 experiments per cada quantitat de llibres per poder fer la mitjana del temps, i per tant, realitzar un experiment de forma més correcta ja que així podem evitar valors *outliers* i al tenir sempre la mateixa llavor es pot replicar.

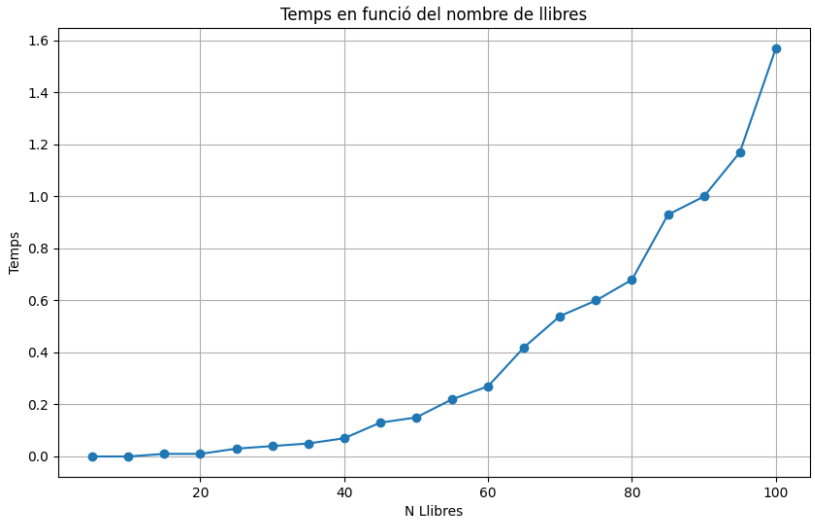

Figura 5: Gràfic sobre el temps d'execució respecte el nombre de llibres

Observant els resultat obtinguts (veure 5), podem afirmar que la nostra hipòtesi era certa ja que podem observem un patró exponencial. A més, caldria mencionar que el codi funciona ja que no s'ha creat mai una seqüència de més de 12 llibres predecessors, la qual cosa no seria resoluble però podria passar.

Pel que fa a l'extensió 1, el procediment que vam realitzar va ser el mateix que en la bàsica i les conclusions que podem observar són molt similars. Com es pot veure en la figura 6 el creixement també és exponencial i, al comparar els temps de les dues extensions es fàcil veure que aquesta última és bastant més demandant pel simple fet de que de mitjana triga 10 vegades més.

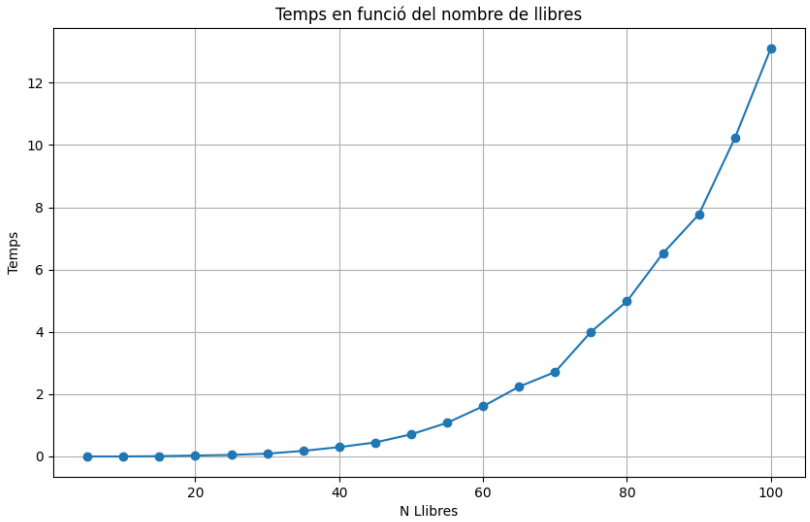

Figura 6: Gràfic sobre el temps d'execució respecte el nombre de llibres

Aquest experiment es va començar a complicar una vegada vam avançar d'extensions i volíem començar a introduir els paral·lels. Una vegada portavem unes quantes execucions ens vam adonar que el temps no era proporcional al número de pàgines sinó a la dificultat en si del propi problema plantejat. Això dificultava molt el seu anàlisi i, per tant no vam poder afegir al *dataset* la informació de les dues últimes extensions en aquest experiment.

En algunes llavors aleatòries, 25 llibres podria tardar 0.01 segons mentre que en altres amb tan sols 15 ja rondava els 5 minuts. A banda d'això a vegades un problema tardava molt temps però pel simple fet d'eliminar una aresta que relacionava dos llibres automàticament es resolia instantàniament. Aquest problema es pot observar i llegir de forma més detallada en la Figura 12.

4.1.2 Segon experiment

En aquest segon experiment volem tractar i analitzar la diferència que hi ha si mantenim constant el nombre de llibres però, enlloc de tenir una única cadena molt llarga de llibres predecessors o paral·lels, en tenim unes quantes de més curtes. Per poder realitzar aquest experiment, primer ens centrarem només amb els llibres predecessors i posteriorment amb els llibres paral·lels, ja que probablement obtindrem resultats diferents depenent del tipus de relació entre llibres. La nostra hipòtesi inicial és que el fet de tenir una cadena més llarga resultarà en una complexitat molt major degut a que, per exemple, en el cas de que els llibres siguin predecessors, haurà de canviar el mes menys vegades i podrà planificar més fàcilment tots els llibres en menys mesos.

La metodologia per crear la base de dades de l'experiment va ser, en el cas dels predecessors, crear una cadena inicialment de 10 i anar augmentant de 10 en 10 fins arribar a 100. En el cas dels llibres paral·lels, degut a la seva major complexitat vam decidir començar en un grup de 2 llibres i anar

augmentant una unitat fins arribar a 10 on ja es podia veure clarament la tendència.

Els dominis utilitzats han estat el bàsic i la extensió 2 per els llibres predecessors i paral·lels respectivament. Això és degut a que en aquest experiment no volíem influència del nombre de pàgines i volíem que en el cas dels predecessors es comportés més ràpid comparat amb si utilitzessim una altra extensió on hi hauria més predicats i condicions que no afecten al nostre problema però augmenta la complexitat.

Aquesta base de dades consta de diferents elements com ara el nombre de llibres, els estats visitats, la profunditat màxima que ha hagut de fer el planificador, el nombre d'accions *easy* i *hard* alhora que el temps que ha tardat en cada observació; tota aquesta informació provenia de la pròpia sortida del planificador.

Vam procedir a generar diferents gràfiques per tal de poder comparar correctament els tipus de llibres a mesura que la complexitat augmentava i posteriorment realitzar un anàlisi més profund amb tota la informació.

4.1.2.1 Estats

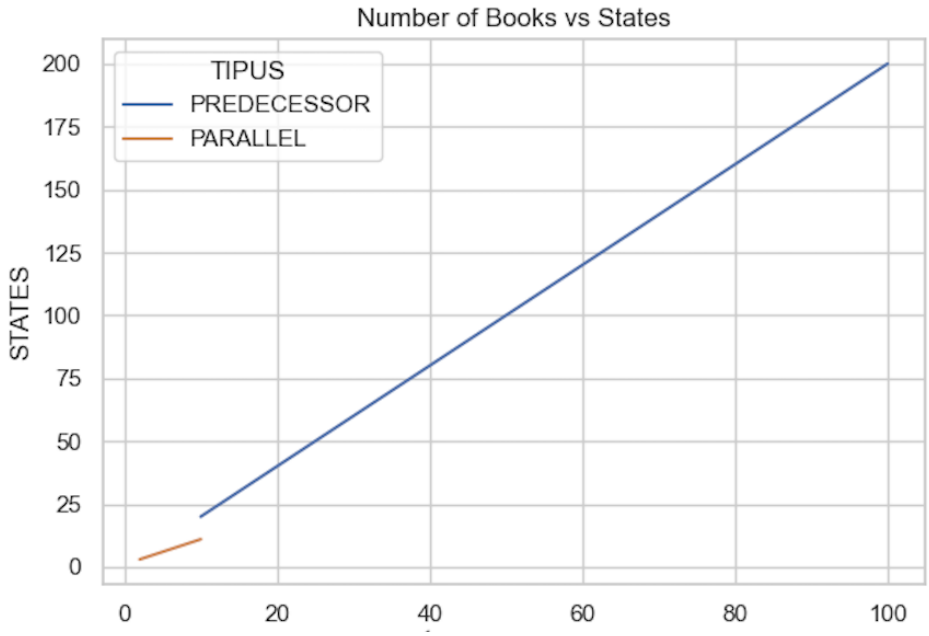

Figura 7: Imatge sobre l'augment del nombre d'estats respecte a la longitud del grup de llibres

Si analitzem les dades obtingudes de totes les cadenes de predecessors podem treure informació molt interessant. A primera vista destaca el creixement lineal tant en els paral·lels com en els

predecessors, tal i com indica la Figura 7. Més concretament és de 2n pels predecessors i de n+1 per als paral·lels (sent n la longitud màxima de la cadena de llibres).

En el cas dels predecessors, cada llibre afegit a la cadena introdueix un nou conjunt d'estats associats amb la seva lectura. Com que cada llibre té un predecessor, la relació és directa i lineal, duplicant efectivament el nombre d'estats necessaris; d'aquí és d'on surt aquesta relació en la gràfica.

En el cas dels llibres paral·lels, cada llibre afegit incrementa els estats, però no de manera tan dràstica com en el cas dels predecessors. Això és degut a la independència dels llibres paral·lels entre si segons la nostra manera de relacionar-los on, només estan connectats a un. Cada llibre nou, per tant, afegeix un únic estat addicional, més un estat base comú, resultant així els n+1 estats.

4.1.2.2 Accions fàcils

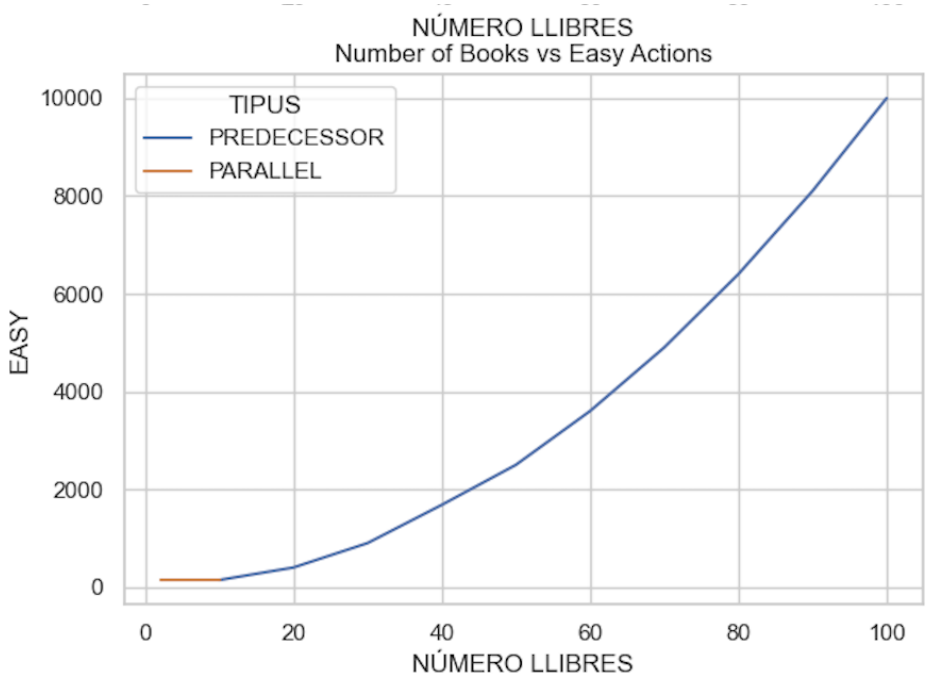

Figura 8: Imatge sobre l'augment de les accions fàcils respecte a la longitud del grup de llibres

A simple vista es pot veure en la Figura 8 que en el cas dels paral·lels el nombre d'accions fàcils no augmenten però en el cas dels predecessors ho fa de forma bastant lineal.

En el nostre domini una acció fàcil podria ser assignar un llibre a un mes, en el cas que aquest no tingui dependències, o marcar un llibre com a llegit.

En el cas dels predecessors es pot atribuir aquest creixement a operacions com assignar la lectura d'un llibre, que es realitzen independentment per a cada llibre. Com més llarga sigui aquesta cadena, més accions fàcil d'assignacions requereixen.

En els llibres paral·lels les accions fàcils romanen constants independentment del nombre de llibres, això és degut a que malgrat que la complexitat general augmenti molt, la complexitat individual no ho fa i les accions fàcils, per tant, tampoc.

4.1.2.3 Accions difícils

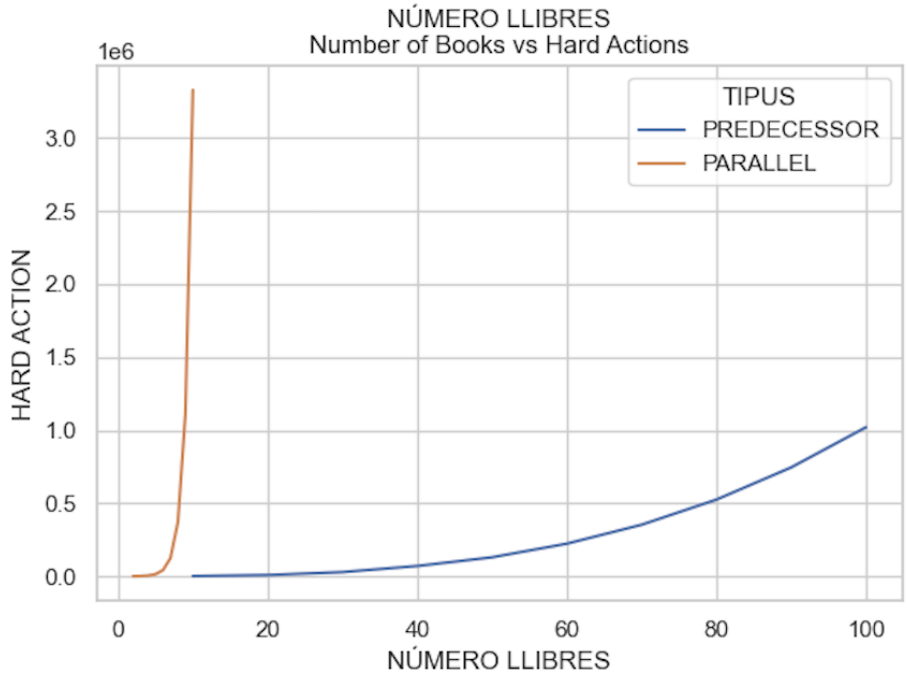

Figura 9: Imatge sobre l'augment de les accions difícils respecte a la longitud del grup llibres

Pel que fa a les accions difícils, tal i com es pot veure en la Figura 9, en el cas dels llibres paral·lels creixen de forma molt exponencial però en el cas dels predecessors de forma més suau, apropant-se a una relació lineal.

Les accions difícils (*hard*) són aquelles que demanen i involucren una major complexitat, ja que requereixen la consideració de múltiples factors, dependències o restriccions. En el nostre domini, aquestes podrien incloure planificar la seqüència de lectura en funció de dependències o inclús gestionar el número límit de pàgines mensuals.

En el cas dels llibres predecessors, l'augment d'accions difícils és més gradual ja que cada llibre

predecessor afegit introdueix noves restriccions i dependències, però d'una manera lineal, ja que només depèn del seu predecessor immediat.

Això no es compleix en el cas dels llibres paral·lels , això és degut a la complexitat de coordinar diversos llibres independents dins del mateix marc temporal, especialment tenint en compte el límit de pàgines mensuals (encara que en aquest experiment no les hem considerades). A mesura que s'afegeixen més llibres paral·lels, la complexitat de planificar la seva lectura dins dels límits establerts augmenta de manera molt exponencial.

4.1.2.4 Profunditat màxima

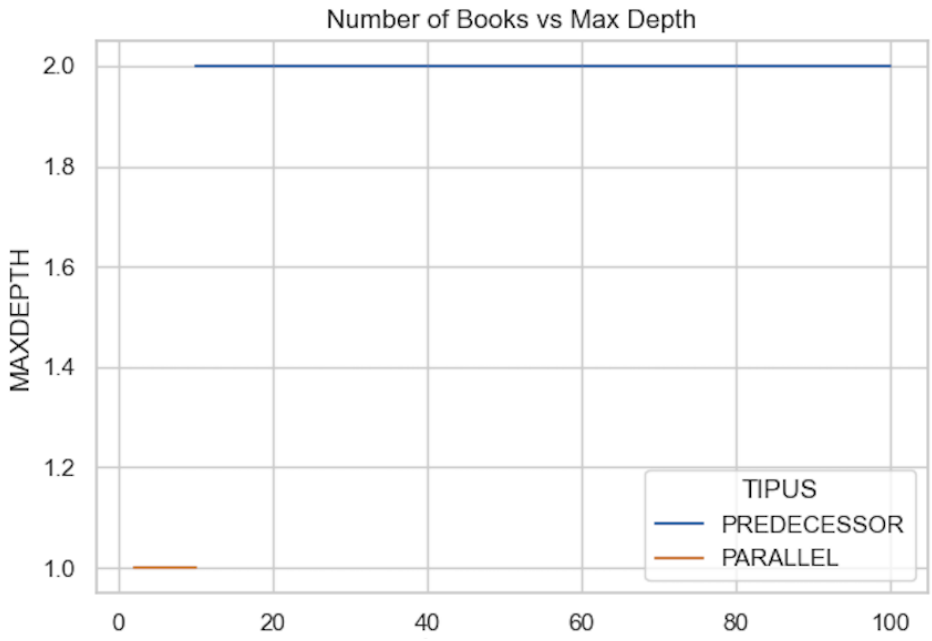

Figura 10: Imatge sobre la profunditat màxima respecte a la longitud del grup de llibres

A l'analitzar la Figura 10, es pot observar que en cap dels casos la profunditat màxima varia. Això realment no es sorprenent per la definició en com estan representats els predicats.

En el cas dels predecessors, la profunditat màxima sempre és 2 i, de fet, en cada iteració de cada execució és 2 excepte en la última on troba la solució, degut a que cada llibtre té una dependència directa del seu predecessor, formant una cadena amb una profunditat de dos nivells.

En el cas dels paral·lels això no passa ja que, com ja hem mencionat anteriorment, es comporten de forma individual i no hi ha dependències seqüencials entre els llibres; tot això fa que la profunditat

màxima sigui de 1 en tots els casos.

4.1.2.5 Temps

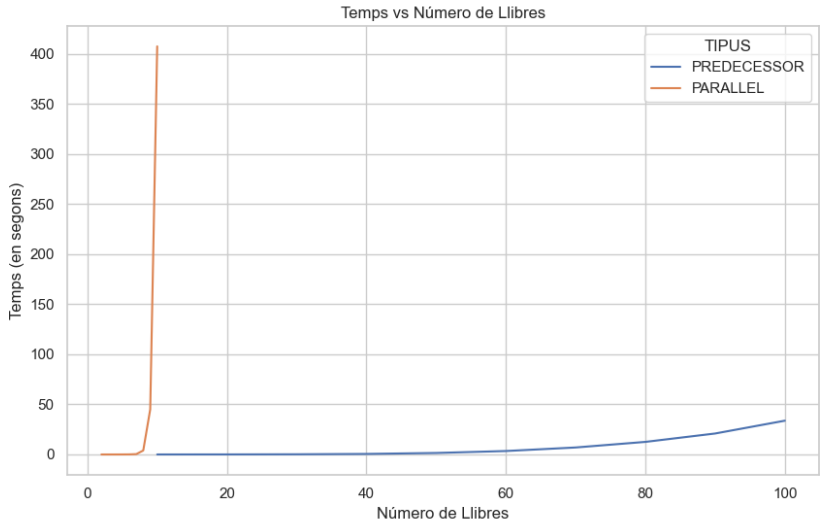

Figura 11: Imatge sobre el temps d'execució respecte a la longitud del grup de llibres

En aquest últim experiment es pot observar que l'evolució del temps en el cas dels paral·lels és gairebé una línia vertical.

En el cas dels llibres predecessors, podem trobar un augment progressiu i relativament suau a mesura que el número de pàgines augmenta, aquesta tendència indica que, mentre més llarga és la cadena de predecessors, més temps necessita el planificador per trobar una solució. Això és deu, molt probablement, a la dependència seqüencial adicional que comporta afegir una nova relació. Si comparem aquest experiment amb els resultats de l'experiment 4.1.1 es veu una clara diferència ja que en el cas de tenir 100 llibres cadascun predecessor d'un altre però sense ser una cadena unida el temps total és d'aproximadament 1.57 segons respecte els 33.73 que suposo si tot és una única cadena.

En el cas dels llibres paral·lels es pot veure que, encara que fins als primers 7 llibres ho fa gairbé instantàniament, només cal afegir-ne 3 més per que trigui gairebé 7 minuts. Això resulta molt interessant i és degut al fet que les accions difícils també augmenten de forma exponencial com ja s'ha vist en la Figura 9.

4.1.2.6 Conclusions Finals

Les conclusions globals de l'experiment suggereixen que, mentre que la planificació de cadenes de predecessors pot ser escalada de manera eficient amb els algoritmes actuals, la planificació de llibres paral·lels requereix una atenció particular, especialment més enllà del punt d'inflexió que es troba a l'afegir el vuitè llibre. Aquests resultats destaquen la importància de dissenyar planificadors PDDL que puguin adaptar-se dinàmicament a l'augment de complexitat i implementar estratègies d'optimització que puguin anticipar i superar punts de congestió dins del procés de planificació.

Els resultats també podrien informar el desenvolupament de mètodes de descomposició de problemes o l'aplicació de restriccions addicionals que puguin simplificar l'espai de cerca sense comprometre els objectius de planificació. Entendre bé com es comporta el programa de planificació quan està realitzant una tasca amb un cost computacional elevat és molt important. Això ens ajuda a fer que aquests programes funcionin millor en situacions reals, on cal que siguin ràpids i eficients.

4.1.3 Tercer experiment

En aquest experiment volem analitzar com varien el temps i la diferència d'estats visitats si apliquem l'optimització que busca minimitzar el nombre de mesos creats (flag `-O` en la comanda d'execució). La nostra hipòtesi inicial és que augmentarà molt la complexitat, probablement de manera exponencial, alhora que el temps d'execució; ja que, no només caldrà una solució vàlida per planificar els llibres, sinó que haurà de ser, també, la més òptima. Per tal d'exprimir el màxim l'optimitzador utilitzarem l'extensió 3 per tal de tenir en compte pàgines als llibres.

Si ens posem a pensar en aquest problema, resulta molt similar al famós problema del *empaquetament binari* (*Bin Packing*) [2] d'optimització combinatòria. És fàcil fer la comparació, ja que només cal substituir els mesos pels contenidors les 800 pàgines màximes que es poden llegir cada mes per l'altura d'aquests contenidors i els llibres que calen llegir pels paquets que es s'hi assignen. Aquest problema té una dificultat NP-difícil [4], sent així un problema sense algorisme conegut per tal de trobar la millor solució en un temps polinòmic i, per tant, sense necessitat de realitzar l'experiment ja ens podem fer la idea de que la nostra hipòtesi inicial serà certa.

Per tal de poder avaluar com funciona aquest optimitzador i degut a la seva gran complexitat farem dos avaluacions. En la primera es plantejarà un cas normal amb 9 llibres, tots objectiu, que contenen alguna relació entre si i es compararà sense l'optimitzador per tal d'observar si realment influeix en la reducció dels mesos el fet de tenir aquest minimitzador. D'altra banda, la segona prova consistirà de simplement els mateixos 9 llibres amb les mateixes pàgines però aquesta vegada sense tenir cap relació entre si. Això, efectivament, equival al problema NP mencionat anteriorment del *Bin Packing*, ja que resulta molt interessant comparar si el fet de tenir relacions entre si simplifica i retalla l'espai.

En la primera prova sense l'optimitzador ha tardat un total de 0.01 segons amb un nombre de 26 nodes creats arribant fins al mes de setembre i amb l'optimitzador l'execució ha tingut una durada de 0.83 segons amb 16688 nodes creats però es redueix tot a tan sols 7 mesos, és a dir fins al juliol. Com es pot veure en aquest cas hi ha una diferència molt clara ja que tot està reordenat guanyant així 1 mes però també requereix un cost computacional bestial ja que mentre un pràcticament és instantani l'altre ha tardat gairebé un segon; cosa que no està malament, però com que un és 0.01 és un canvi molt significatiu.

En la segona prova, sense l'ús de l'optimitzador, el procés ha durat un total de 0.01 segons, ge-

nerant 53 nodes i arribant fins al mes de juliol. Amb l'optimitzador, en canvi, la durada ha estat de 430.2 segons, amb 383226 nodes creats, però s'ha restringit a 6 mesos, és a dir, fins al mes de juny. Aquesta comparació revela la diferència d'1 mes. No obstant això, implica un augment considerable en la càrrega computacional: mentre que una execució és gairebé immediata, l'altra requereix aproximadament 43020 vegades més temps.

En conclusió com es pot observar l'omptimitzador realment funciona i ens és útil per tal de trobar la millor solució, però, cal tenir en compte que no és gens aplicable a la vida real ja que un usuari no pot estar esperant tant de temps per tenir la millor resposta sinó que és molt millor donar una resposta aproximada amb un temps eficaç. S'ha de mencionar que no és gens escalable i, a mesura que augmentem els llibres, tardarà més i més de forma totalment exponencial com passa en el problema *Bin Packing* on si tinguessim n paquets i 12 mesos realment hi hauria una complexitat de l'ordre de 12^n (realment menys degut a totes les restriccions reals com ara que no caben tots els paquets en un contenidor) que faria totalment impossible portar a la pràctica aquest optimitzador. A més a més cal tenir en compte també que el propi planificador crea una heurística que molt probablement no serà la més eficient per aquest cas.

També s'ha de mencionar, com ja havíem fet bé en les hipòtesis prèvies a l'execució que, efectivament, el temps es troba molt reduït en el cas que afegim relacions entre els propis llibres degut a que obliga a col·locar llibres en diferents mesos i, per tant, cal comprovar menys combinacions que en el cas en que no en disposem. Això a primera vista pot semblar bastant contradictori que les relacions puguin ajudar a trobar la millor solució però es pot veure bastant simple si pensem en una cadena de k elements predecessors, la única solució optimitzada serà un llibre en cada mes ocupant així un total de k mesos. En canvi, si tenim els k llibres sols, haurà d'intentar provar totes les combinacions per assegurar-se d'aquest valor òptim cosa que el tornarà en molt ineficient.

4.2 Anàlisi de la complexitat

Una vegada realitzats tots els experiments, podem començar a analitzar com d'òptima és la nostra implementació tenint en compte el nombre d'estats visitats i el temps d'execució, entre altres factors.

Hem pogut veure en els resultats obtinguts certs patrons significatius en la relació entre la complexitat dels problemes de planificació i l'estructura dels llibres involucrats, sent aquests cadenes de predecesors o grups de llibres paral·lels.

Per a les cadenes de predecesors, s'ha observat un creixement lineal en el nombre d'estats visitats i en el temps d'execució, la qual cosa implica que l'augment de la longitud de la cadena no introdueix una complexitat desmesurada, mantenint un comportament previsible i gestionable pels algoritmes. Per altra banda, els grups de llibres paral·lels han mostrat una complexitat exponencial, especialment després d'arribar a un punt d'inflexió en afegir el vuitè llibre, moment en el qual la càrrega computacional es veu incrementada dràsticament.

L'optimització per minimitzar el nombre de mesos creats (funció `num_months_created`) ha demostrat ser bastant efectiu per tal reduir la durada dels plans, que era l'objectiu pel qual l'implementàvem, però realment ha comportat un augment massa significatiu en el temps de càlcul, lo qual provoca que aquest intent d'optimització sigui massa ineficient, doncs a la realitat és important que el temps de càlcul no sigui extremadament elevat. Per tant, la nostra conclusió és que és una bona opció tractar de minimitzar aquesta mètrica en problemes poc complexos i senzills, però en els que ja requereixen d'una certa complexitat un tant elevada, no surt gens a compte aquesta optimització de mesos creats.

En resum, la complexitat en aquesta pràctica ha mostrat una gestió eficient en el cas de cadenes de predecesors, presentant un comportament lineal tant en el nombre d'estats visitats com en el temps d'execució. Tot i això, les configuracions amb grups de llibres paral·lels presenten una complexitat exponencial, especialment després d'arribar a un punt crític amb l'addició del vuitè llibre. Això ens indica que s'haurien d'explorar noves estratègies en el disseny de planificadors PDDL, per tal d'afrontar amb èxit escenaris de major complexitat i assegurar el desenvolupament de solucions més eficients i escalables.

4.3 Problemes confrontats en les execucions

En l'execució dels experiments (o de jocs de prova individuals), de tant en tant en hem trobat amb un problema. El patró era comú, realitzar lleugers canvis a un problema resultava amb una augment de la complexitat brutal, això però era molt difícil d'analitzar.

Com ja s'ha vist en l'experiment (4.1.1), el fet de tenir llibres paral·lels a vegades implicava crear relacions molt difícils pel planificador per poder solucionar-ho. Això sobretot es notava quan un llibre era paral·lel d'uns altres i just també tenia predecessors alhora que ell mateix era predecessor d'altres llibres. Encara que aquest problema tingués molt poques relacions entre els llibres en si, resultava molt més costós computacionalment que un problema amb 10 vegades més de llibres i 10 vegades més d'arestes però sense aquesta complicació.

Una altra vegada, això podem deduir que és degut al propi planificador com ja passava a l'experiment 3 (4.1.3)on deduïem que la funció heurística generada no deuria ser suficientment bona per

tal d'afrontar algun problema particular.

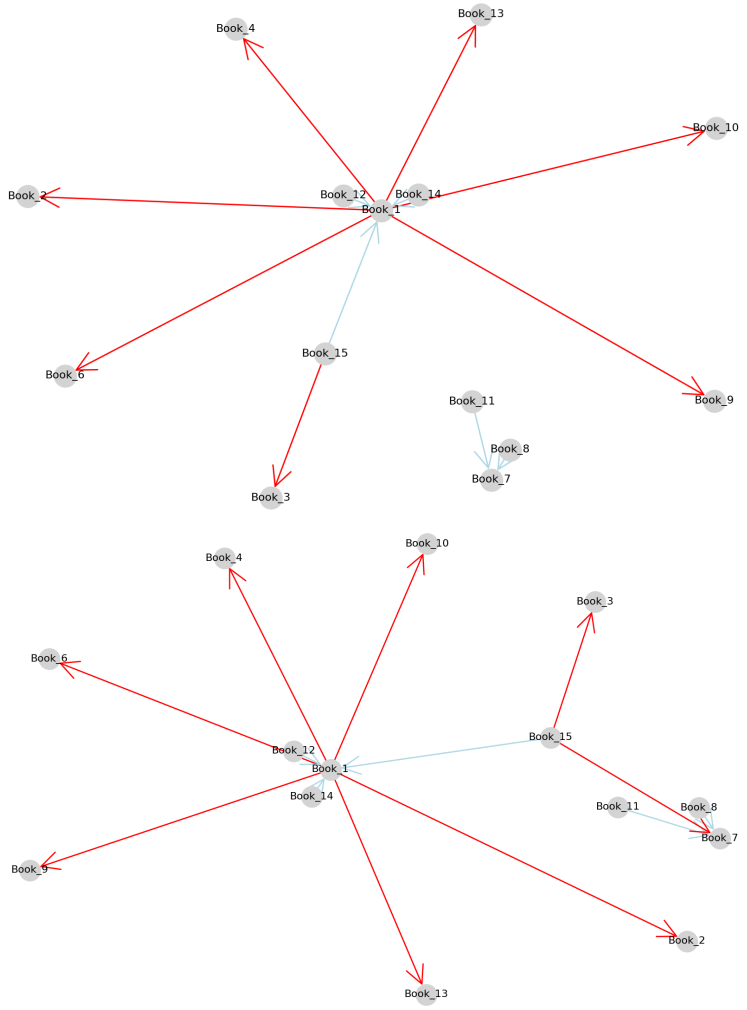

Figura 12: Visualització de dos jocs de prova molt similars, però amb temps d'execució radicalment diferents. En la imatge superior, es veu un joc de proves d'extensió 2 executat en 0.44s. En la imatge inferior, es veu el mateix joc de proves, però amb l'addició de l'aresta $Book_15 \rightarrow Book_7$ (de tipus *parallel*), que comporta un temps d'execució de 105s.

Com es pot veure en aquesta figura 12 encara que només afegim una aresta la complexitat és multiplica per 200. A l'analitzar millor que està passant realment, es pot veure que encara que els dos problemes són resolubles en un tenim dos elements connexos (arbres) i en l'altre en tenim tan sols un. Com ja s'ha pogut comprovar també en experiments com l'experiment dos (4.1.2) el fet de tenir un grup de llibres més gran augmenta dràsticament el temps d'execució a banda de les accions difícils del planificador i, justament, en aquest exemple l'aresta pont entre aquests dos components

separats és paral·lela.

D'aquí es poden treure les conclusions que ens feien falta de l'experiment 1 (4.1.1) on mencionavem que no es podia mesurar el temps en el cas de les extensions 2 i 3 degut a que no era proporcional al nombre de llibres.

5 Conclusions

Un cop finalitzat el treball, podem concloure que l'ús de PDDL, malgrat desagradar-nos en un principi, ens ha permès realitzar una representació 'simple' i clara del problema, facilitant la comprensió de la lògica i les restriccions especificades a l'enunciat. Això ha estat clau per assegurar l'exactitud dels resultats i la consistència en la generació de plans de lectura. Per altra banda, hem de destacar destacar la creació del generador de jocs de prova en Python, ja que es tracta d'una eina molt útil per validar el domini i el problema de manera ràpida i eficient. Cal remarcar que és de les coses que més feina ens ha portat, ja que és on hi hem dedicat més hores, tan solucionant errors de codi com tractant de que sempre generés arxius de problema que tinguessin un sentit lògic i sintàctic, i així ens permetessin validar els resultats i analitzar-los. En acabar la pràctica, podem afirmar que, satisfactòriament, hem complert amb el que ens proposàvem.

Una altra cosa a comentar és que, tot i que inicialment semblava molt complex, l'ús de fluents i la seva implementació en el nostre programa ens ha resultat molt més fàcil del que pensàvem. Mentre que altres aspectes del codi ens han consumit més temps del que ens hauria agradat, les funcions han resultat ser un aspecte bastant simple, ja que un cop vam entendre del tot com funcionen els fluents, la seva implementació va estar prou senzilla. En contraposició, una de les majors dificultats que s'ens ha presentat en el treball ha estat el fet de plantejar com fer el generador, ja que havíem de crear un arxiu que permetés crear jocs de proves amb totes les restriccions que conformen un problema resoluble, i això inicialment ens va suposar certs molts esforços.

També ens ha sigut molt útil el fet de realitar diversos experiments, on hem pogut extreure diverses conclusions i analitzar més en profunditat el problema en qüestió. Concretament, una de les principals observacions que realitzat és la relació directa que hi ha entre el temps d'execució del programa i la mida de l'input (nombre de llibres, longitud de les cadenes de predecessors o quantitat de llibres paral·lels). Com més gran és el conjunt de dades d'entrada, més temps necessita el planificador per arribar a una solució. A més, aquesta tendència es fa clarament visible en el cas dels llibres paral·lels, ja que com hem pogut veure en l'experiment 2, el fet d'afegir els predicats `parallel` fa que augmenti de manera molt significativa el temps d'execució del programa, la qual cosa fa que sigui molt poc escalable, ja que a mesura que augmentem la complexitat del mateix, el temps d'execució creix exponencialment, lo qual provoca que es trigui massa en executar-lo.

Per tant, si comparem les cadenes de llibres predecessors amb les cadenes de llibres paral·lels, veiem que la complexitat de les primeres augmenta de forma lineal, mentre que la de les segones ho fa de forma exponencial, fet que ens ha indicat que era molt més ràpid fer jocs de prova amb més llibres predecessors que paral·lels, ja que si ho fèiem al revés, hauríem trigat molt més temps en executar els programes.

Al realitzar els experiments també ens hem adonat del gran augment en quant a la complexitat del problema quan s'introdueix un objectiu d'optimització, en què el fet d'afegir restriccions d'optimització tendeix a augmentar considerablement la dificultat del problema, ja que hem vist com el programa trigava excessivament més temps en trobar la solució.

Cal remarcar, per tant, que els resultats obtinguts al llarg del treball també permeten extreure algunes conclusions sobre l'eficiència dels algoritmes utilitzats en el nostre planificador, ja que tot i haver demostrat que són eficaços en escenaris simples o amb una complexitat moderada, quan ho hem escalat a situacions amb una gran complexitat (com en el cas dels llibres paral·lels o amb

l'optimitzador activat), l'eficiència dels algoritmes ha disminuït clarament.

5.1 Valoració de l'aprenentatge adquirit

Aquest pràctica ens ha servit majoritàriament per entendre com funciona el llenguatge PDDL en detall i aprendre a utilitzar-lo correctament, és a dir, a com modelar les diferents característiques del domini i el problema per tal que el planificador funcionés correctament. A més, aquest treball ha enfortit els nostres coneixements sobre python, ja que els arxius *.py* que hem creat també ens han suposat tot un repte, doncs des d'un inici volíem fer un generador interactiu, i vam decidir que la millor opció era que es basés en una estructura de grafs, la qual cosa no havíem aprofundit fins al moment, ja que únicament l'havíem estudiat en l'assignatura del Q1 anomenada Fonaments Matemàtics, però sense arribar a implementar un codi corresponent. Un cop acabat el treball, podem afirmar que, satisfactòriament, hem après molt sobre aquest llenguatge, tenint en compte que abans de començar amb la pràctica els nostres coneixements sobre PDDL eren pràcticament nuls.

6 Referències

[1] Wikipedia contributors. Knapsack problem — Wikipedia, the free encyclopedia, 2023. [Online; accessed 3-December-2023].

[2] Wikipedia contributors. Bin packing problem — Wikipedia, the free encyclopedia, 2023. [Online; accessed 3-December-2023].

[3] Wikipedia contributors. Np-completeness — Wikipedia, the free encyclopedia, 2023. [Online; accessed 3-December-2023].

[4] Wikipedia contributors. Np-hardness — Wikipedia, the free encyclopedia, 2023. [Online; accessed 3-December-2023].