
# Ingeniería Informática, 4º Curso
# Ingeniería del Conocimiento
# Práctica 2: Planificación automática

García García, Alba María 100346091
Martínez Castillo, Irene 100346051
Leganés Grupo 83

29 Diciembre 2018

# Índice

# 1. Introducción

En este documento se explica la **implementación de un planificador automático capaz de realizar partidas** de una versión muy simplificada **del juego de mesa *Agricola*** en su versión familiar y para dos jugadores, mediante PDDL *(Planning Domain Definition Language)*.

La **estructura** de este documento se divide en: un manual técnico donde se explica en detalle las decisiones de diseño tomadas y una descripción detallada del dominio implementado; después, una descripción de las pruebas realizadas para la comprobación del correcto funcionamiento del sistema; y un último apartado con conclusiones técnicas y comentarios personales.

# 2. Manual técnico

Hemos dividido el manual técnico en dos secciones fundamentales: las decisiones de diseño tomadas debido a las limitaciones del planificador *Metric-FF* y la descripción del dominio de nuestro planificador. La primera sección estará marcada por las diferentes versiones por las que ha pasado nuestra implementación y las restricciones y alcance del mismo; mientras que en la segunda se explicarán cada uno de los elementos que componen el dominio (taxonomía de objetos, predicados, funciones, métrica y acciones).

## 2.1. Decisiones de diseño

### 2.1.1. Evolución de las versiones implementadas

Nuestro **objetivo** en esta práctica era implementar un simulador del juego *Agricola* que tuviera **todas las acciones** mostradas en la primera práctica (a excepción de aquellas que se pedía expresamente no incluir en el enunciado) y una vez conseguido esto, añadirle toda la **complejidad y fidelidad al juego original que permitía el planificador** en un tiempo razonable (alrededor de media hora, el tiempo estimado por partida). Teniendo en cuenta estos puntos, diseñamos **tres versiones** del juego:

- **Versión 0**. Esta primera versión contaba exclusivamente con la estructura básica del simulador del juego, es decir, las acciones `jugar`, `cambiar-jugador`, `cambiar-turno` y `terminar`. Por tanto, no se había implementado ninguna acción específica, no estaba desarrollada la taxonomía de objetos y se utilizaban apenas unos pocos predicados. Sin embargo, sí que simulaba el comportamiento de dos jugadores, con sus respectivos cambios de turno. El planificador no mostraba tener ningún problema para generar un plan con esta versión, siendo sus soluciones siempre instantáneas.

- **Versión 1**. En esta segunda versión se incluyeron todas las acciones que formaban parte de la primera práctica, aunque de forma esencial; es decir, simplificando la estrategia de modo que todas las acciones se pudieran realizar de una única forma independientemente del estado del juego. ¿A qué nos referimos con esto? Si tomamos como ejemplo la acción `construir-vallas`, vemos que en CLIPS el jugador construía tantas vallas como su número de maderas y el máximo número de vallas por jugador le permitiera; sin embargo, en esta versión en PDDL, el jugador siempre construirá cuatro vallas siempre que no exceda el máximo permitido, de modo que el pasto objetivo quede totalmente vallado (no se han incluido pastos dobles como sucedía en la primera práctica). ¿Por qué tomamos esta decisión? Nos dimos cuenta que cuantas más acciones incluíamos, más tiempo tardaba el planificador en generar un plan. Como sabíamos que este tiempo crece exponencialmente con la métrica, la única licencia que nos tomamos en cuanto a las acciones implementadas fue generar dos versiones de la misma en aquellos casos que lo exigía la funcionalidad mínima que queríamos introducir (por ejemplo, el caso de `ampliar-familia-con-habitacion` y `ampliar-familia-sin-habitacion`). Sin embargo,

esta versión presentaba un problema grave, y es que el número de acciones disponibles estaba reducido a las implementadas, por lo que no le podíamos pedir a los jugadores que no repitieran acciones, ya que se quedaban sin opciones disponibles. Esto llevaba a que todos los jugadores repitieran las mismas acciones en el mismo orden en cada turno, lo que no resultaba interesante.

- **Versión 1.5**. Nuestra última versión contempla las cartas que proporciona el juego a los jugadores para poder realizar las distintas acciones dentro de la fase de `JORNADA LABORAL`. Las únicas cartas añadidas en la versión anterior eran adquisiciones mayores, de modo que se pudieran realizar las acciones de `obtener-adquisicion-mayor`. Una vez realizado este cambio, y al tener cada acción distintas cartas asociadas, el abanico de acciones creció significativamente. Con esto, ya podíamos aplicar la restricción del juego original en que una misma carta sólo se puede utilizar una vez por turno. Con este fin creamos el predicado `usada ?jugador ?carta`. Ahora, cada vez que probábamos esta versión, veíamos como ambos jugadores realizaban las mismas acciones cada turno en el mismo orden (utilizando cartas distintas, por supuesto). Una vez que vimos que el comportamiento era el adecuado, incluimos la métrica, `total-cost`, ya que considerábamos que la funcionalidad de esta versión era bastante completa y que sólo en el caso de llevar muy bien los tiempos de ejecución se podría añadir algo más. Sin embargo, el tiempo de ejecución se disparó tras utilizar la opción de optimización (las ejecuciones duraban más de una hora). En un principio, creímos que esto se debía al amplio abanico de acciones que el planificador tenía disponible en cada elección; pero resultó no ser así tras hacer distintas pruebas. Por tanto, pudimos mantener la funcionalidad completa que deseábamos en un principio aplicando distintos retoques que veremos a continuación.

### 2.1.2. Mejoras añadidas a la última versión estable

Aplicamos distintas **mejoras para poder conseguir la meta de una ejecución que no superara los 30 minutos sin perder la funcionalidad mínima contemplada** (dichas pruebas se realizaron usando tres turnos y dos personas por jugador):

- **En un principio habíamos evitado las funciones a toda costa**, ya que se nos explicó en clase que estas tenían peor rendimiento que los predicados. Así, en la versión 1.5 inicial de nuestra implementación, tanto el cambio de turno como el contador de acciones realizadas por jugador se realizaban mediante el predicado `next ?contador1 ?contador2`, utilizando un objeto de tipo `contador` para ello. Siguiendo esta recomendación, buscamos cambiar todas aquellas funciones que sumaban valores constantes por sus predicados homólogos (no se pueden sumar valores que dependan del estado de juego con el predicado `next`). Sin embargo, la única que cumplía estos requisitos era la función `n-vallas`, por lo que su impacto en el tiempo de ejecución no podía ser muy significativo. Tras probarlo y ver que el problema seguía sin solucionarse, **probamos a cambiar todos los predicados que usaban `next` por funciones**. Haciendo esto también fue imposible conseguir que ejecutara un plan para tres turnos y dos personas por jugador en un tiempo razonable; pero sí que notamos cambios significativos si bajábamos el número de turnos a dos. Así, y a pesar de que el tiempo de ejecución se quedaba en torno a los 6 segundos en ambos casos, el número de estados generados por el proceso de búsqueda bajaba más de la mitad usando funciones: de 40.034 a 19.094 estados. Esto nos indicaba inequívocamente que **el planificador funcionaba mejor aumentando el número de funciones**.

- Como el problema seguía sin poder resolverse en un tiempo razonable, barajamos incluir distintas **restricciones** que fueran lo suficientemente **estrictas** como para reducir significativamente el número de estados.

  - **La primera opción fue reducir el número de cartas disponibles en el juego**, de modo que hubiera sólo una carta por cada tipo de acción diferente (por ejemplo, con `jornalero` y `pescar` se puede obtener comida, por lo que al ser redundantes mantuvimos sólo una de las dos). Sin embargo, este cambio tuvo muy poco impacto, ya que no podíamos eliminar muchas cartas. Tras ver estos resultados, decidimos eliminar cartas poco a poco hasta que el planificador pudiera obtener una solución. Este proceso no consiguió resultados satisfactorios, ya que **el número de acciones quedó extremadamente reducido** (menos de la mitad de las acciones disponibles al principio, debido a que había acciones que se habían quedado sin todas sus cartas). Por tanto, **esta restricción quedó totalmente descartada**.
  - **La segunda opción** (y la definitiva) **fue reforzar el alcance de la restricción** `usada`**, de modo que no sólo impidiera utilizar la misma carta a todos jugadores durante el mismo turno; sino durante toda la simulación del juego**. Esto no sólo permitía que el número de acciones disponibles se fuera reduciendo poco a poco, sino también que los jugadores no repitieran las mismas acciones en cada turno. Sin embargo, **el planificador seguía sin poder resolver este problema** para tres turnos y dos jugadores por persona, por lo que no era suficiente.

- La siguiente medida que tomamos tuvo que ver con la métrica utilizada. **Inicialmente, creímos conveniente que todas las acciones definidas aumentaran la métrica**, para asegurarnos del correcto flujo de acciones (`jugar - cambiar-jugador - jugar - cambiar-turno - ... - terminar`). Así, aquellas acciones 'internas' del juego (`cambiar-jugador`, `cambiar-turno` y `terminar`) tenían un coste mínimo (entre 0 y 1) y el resto se movían entre distintos valores con una distancia mínima de 10 para diferenciarse, es decir, su coste mínimo era 11. **Se nos ocurrió que podía ser interesante quitar la métrica de estas acciones 'internas'** y utilizar costes a partir de 1 en el resto de acciones. No sabemos muy bien por qué, pero este fue el cambio definitivo para que nuestro simulador pudiera terminar la ejecución tras 30 minutos en un ordenador personal y 39 utilizando Guernika.

- La última mejora no se debió explícitamente al tiempo de ejecución de las distintas pruebas, sino al **incorrecto funcionamiento del programa**. El problema que encontrábamos en todas las pruebas que realizábamos era que, como **las dos acciones de** `ampliar-familia` tenían tan poco coste (1), siempre que **se realizaban era antes de cambiar de turno o de jugador; en una acción que excedía el número de personas disponibles definidas para el jugador**. ¿Por qué sucedía esto? Porque no comprobábamos en las distintas acciones que podía realizar el jugador si ya había llegado a su máximo de acciones en el turno actual. Por tanto, añadimos esa restricción. Sin embargo, esta solución era insuficiente, ya que algunas de las pruebas llegaban a tardar varias horas. Encontramos entonces que **el problema radicaba en el número de personas inicializadas por jugador, que en ese momento era el máximo posible, 5**. Habíamos hecho esto porque no se pueden crear nuevas instancias durante la ejecución, pero esto multiplicaba el número de estados generados. Por tanto, y tras observar que los jugadores nunca realizaban más de una vez la acción de `ampliar-familia`, **creamos sólo una instancia más de las personas que se les habilitaba en un principio**. Así, en nuestra prueba de referencia de dos personas por jugador y tres turnos, llegamos a obtener resultados mucho menores a los 30 minutos.

### 2.1.3. Restricciones finales de nuestra implementación

Una vez explicada cómo es la versión actual de nuestro simulador de *Agricola*, vamos a listar las distintas **restricciones que posee respecto a la práctica anterior** para poder conocer mejor su alcance:

- **Sólo hemos simulado la fase de `JORNADA LABORAL`.**
- **Todas las cartas que se destapan durante la fase de `PRINCIPIO DE RONDA` están descubiertas y disponibles para todos los jugadores desde el principio**, ya que esta fase no existe. Además, en cada una de las pruebas que propondremos no se van a simular todos los turnos, sino sólo unos pocos, por lo que esto no tiene sentido a menos que se juegue la partida entera.
- **No existen las acciones 'conjuntas' ni 'recursivas'** que implementamos en la primera práctica, ya que implica aumentar el número de acciones del juego, lo que resulta insostenible para *Metric-FF*.
- **Cada una de las cartas disponibles sólo se puede utilizar una vez en toda la partida**.
- **Los jugadores no se intercambian el turno**, sino que aquel que es el jugador inicial comienza, realiza todas sus acciones y luego le sigue el otro jugador. Se hizo esto para simplificar la funcionalidad del simulador, aunque no habría sido complicado que los turnos se alternaran.
- Al no existir la fase de `REPOSICIÓN`, **al ejecutar la acción `obtener-recurso-no-animal`, el jugador siempre recoge cinco unidades del recurso que indique la carta utilizada**.
- **Sólo existen pastos simples**, es decir, compuestos por una única celda. Se han eliminado los pastos dobles de la primera práctica.
- Para simplificar el abanico de opciones que presenta la acción `obtener-recurso-no-animal` y también por no existir la fase de `REPOSICIÓN`; **cuando un jugador realiza esta acción recoge dos animales**, siempre y cuando tenga un pasto vallado vacío. De esta forma, como no existen los establos y todos los pastos son simples; nos aseguraremos que el pasto seleccionado queda completo.
- **Al realizar la acción `vallar`, el jugador siempre coloca cuatro vallas**, por lo que el pasto queda siempre completamente vallado.
- **Cuando el jugador realiza una de las dos acciones disponibles para `ampliar-familia`, obtiene una acción disponible extra en el turno actual**, sin tener que esperar al turno siguiente. Hacer la transición entre recién nacidos y adultos habría requerido nuevas acciones 'internas'.

Las otras **restricciones** que hemos manejado han sido las **impuestas por el enunciado del problema**, y son las siguientes:

- **Las únicas cartas de adquisición mayor que se han considerado son las dos cocinas y los hornos de piedra y adobe**.
- **No existe ninguna acción para construir establos**.
- **No se calcula la puntuación final de los jugadores.**

## 2.2. Descripción del dominio

### 2.2.1. Taxonomía de objetos

En primer lugar, es necesario diseñar e implementar una taxonomía de objetos, puesto que las acciones y predicados van a depender de ella. Dicha taxonomía está basada en la ontología que utilizamos para la primera práctica. En la figura mostrada a continuación se ve el **primer nivel de la jerarquía establecida**.

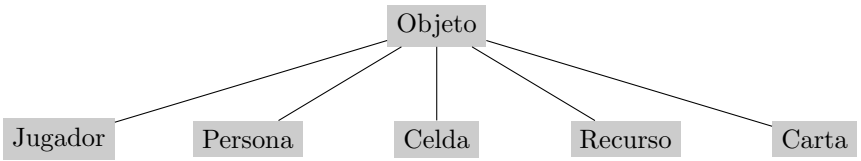


Figura 1: Primer nivel de la taxonomía.

La mayoría de estos objetos cuentan con jerarquías propias que mostraremos a continuación. Lo que representan de cada uno de ellos es la siguiente:

- **Jugador**: Representa a cada uno de los jugadores presentes en el juego. En esta simulación, solamente existen dos jugadores que realizan sus acciones por turnos.
- **Persona**: Representa a cada una de las personas que controla un jugador. El número de personas que controla cada uno de los jugadores se define como parámetro del sistema en el fichero de problema. Todos los jugadores comienzan con una persona más de las que pueden utilizar al principio del juego debido a que en PDDL no se pueden crear nuevas instancias durante la ejecución del programa.
- **Celda**: Representa el conjunto de celdas del tablero de granja de un jugador. Al no incluir restricciones de ortogonalidad ni continuidad del vallado en nuestra versión del juego, sólo necesitamos saber el número de celdas que posee cada jugador de cada tipo (no es relevante conocer en qué posición se ha colocado un campo o un pasto). Dichas celdas pueden estar vacías, tener habitaciones, incluir animales (pastos) o almacenar vegetales (campos). Cada celda solamente puede ser de un tipo.

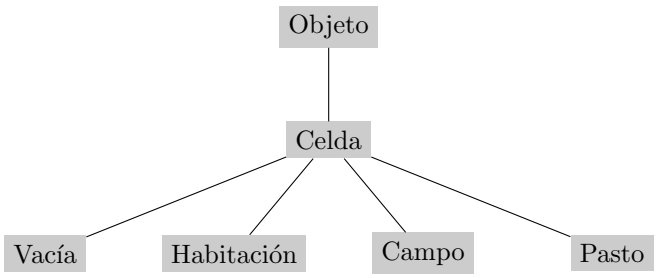


Figura 2: Detalle taxonomía "Celda".

- **Recurso**: Al igual que en el juego original, cada jugador dispone de una variedad de recursos. Para facilitar la implementación en PDDL de este juego, se ha creado la siguiente jerarquía de recursos, que los categoriza según su función. Dicha taxonomía es una de las que barajamos para la primera práctica, pero que desechamos dado que complicaba la generalización de algunas acciones. Sin embargo, la hemos recuperado para esta segunda práctica porque dicha generalización no corre peligro al haber simplificado aún más el funcionamiento del juego.

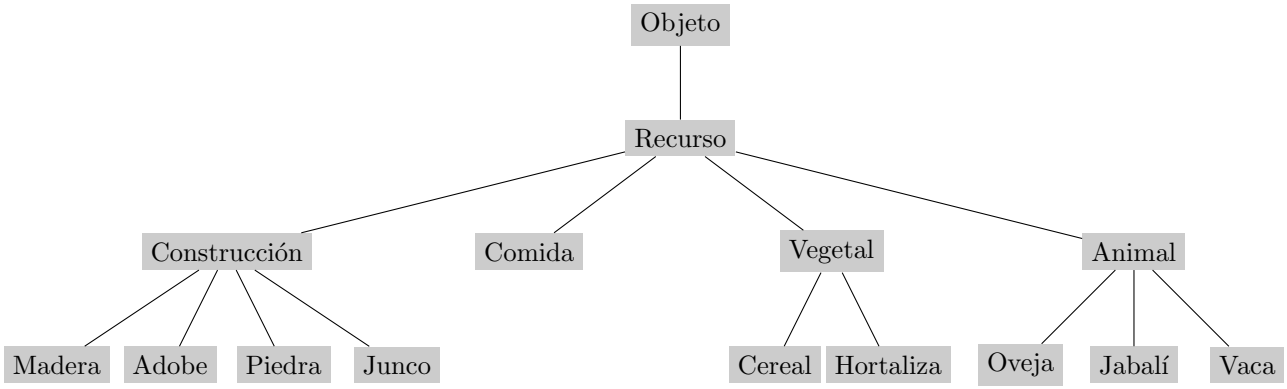


Figura 3: Detalle taxonomía "Recurso".

- **Carta**: Este objeto representa cada una de las cartas que pueden utilizar los jugadores a lo largo del juego. Su jerarquía (figura 4), ha sido simplificada con respecto a la práctica anterior: hemos unificado las "Cartas de tablero" y "Cartas de ronda" en "Cartas de acción" (esta distinción no es necesaria cuando no existe la fase de `PRINCIPIO DE RONDA`) y también hemos eliminado las cartas de mendicidad (sólo se utilizan en la fase de `COSECHA`). La diferencia entre ambas cartas es que las primeras las puede utilizar cualquier jugador en cualquier momento y las segundas se deben adquirir primero para poder ser utilizadas. Dentro de los tipos que se encuentran bajo el denominador de "Cartas de acción", encontramos una jerarquía inexistente en la versión de CLIPS. Dicha taxonomía se empleó para facilitar la clasificación de cada una de las cartas según las distintas acciones que hemos definido, sustituyendo a la jerarquía de acciones que utilizábamos en la primera práctica.

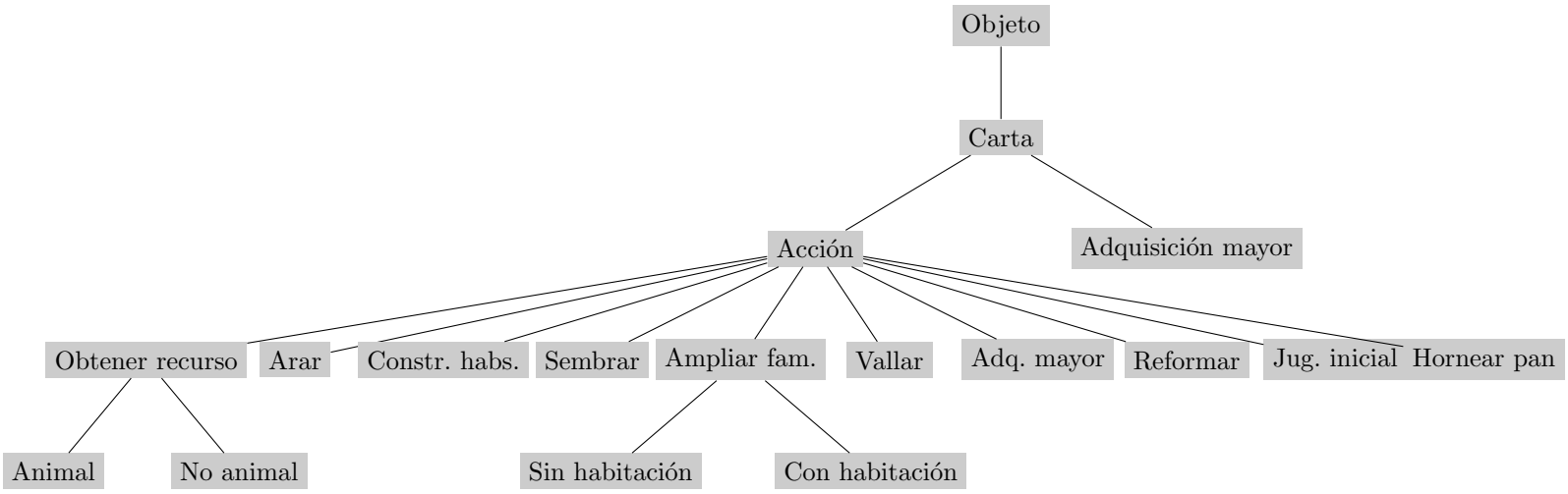


Figura 4: Detalle taxonomía "Carta".

Constr. habs = Construir habitaciones; Ampliar fam. = Ampliar familia;
Adq. mayor = Obtener adquisición mayor; Jug. inicial = Cambiar jugador inicial

### 2.2.2. Predicados

Los **predicados implementados** en el dominio de nuestro problema son los siguientes:

- `(jugador-actual ?j - jugador)`: Determina el jugador ?j que tiene el turno en el momento actual del juego.
- `(pertenece ?p - persona ?j - jugador)`: Indica si la persona ?p pertenece al jugador ?j. De esta forma, cada jugador sólo puede utilizar aquellas personas que tenga asociadas.
- `(recurso-carta ?c - carta_obtener_recurso ?r - recurso)`: Asocia una carta de acción ?c que permite recoger recursos, con el tipo ?r de recurso que provee.
- `(is-animal ?r - recurso)`: Determina si un recurso ?r es de tipo animal. Esta distinción es fundamental para saber qué acción aplicar al utilizar una carta que permite obtener recursos.
- `(material-habitacion ?j - jugador ?r - construccion)`: Establece el material actual ?r de las habitaciones del jugador ?j. Dicho material puede cambiar a través de reformas sucesivas.
- `(transicion-reforma ?j - jugador ?r1 ?r2 - construccion)`: Este predicado determina el recurso necesario ?r2 para reformar las habitaciones con material ?r1. Estas transiciones son madera-adobe o adobe-piedra.
- `(jugador-inicial ?j - jugador)`: Determina qué jugador ?j tiene el título de jugador inicial, es decir, el que empieza la partida en la siguiente ronda.
- `(adq-mayor ?j - jugador ?a - adq_mayor)`: Indica que la adquisición mayor ?a pertenece al jugador ?j. Una adquisición mayor no puede pertenecer a dos jugadores a la vez.
- `(is-adq-mayor-doble ?a - adq_mayor)`: Determina si la adquisición mayor ?a es doble o no. "Doble" significa que el jugador debe aportar dos recursos distintos para obtenerla.
- `(usada ?c - carta_accion)`: Indica si la carta de acción ?c ha sido ya usada por algún jugador durante el juego.
- `(juego-terminado)`: Este predicado se crea cuando termina el juego, es decir, es el criterio de parada. Por tanto, se trata del `goal` (meta) en todos los problemas.

### 2.2.3. Funciones

Las **funciones implementadas** en el dominio de nuestro problema son las siguientes:

- `(turno-actual)`: Contabiliza el turno en el que se encuentra la partida. Su valor se encuentra en el rango [1, `ultimo-turno`].
- `(ultimo-turno)`: Determina el número máximo de turnos a jugar. Se trata de uno de los dos parámetros del programa, junto con el número de personas asociadas a cada jugador.
- `(acciones-realizadas ?j - jugador)`: Indica el número de acciones realizadas por el jugador ?j durante el turno actual. Este número nunca será mayor que el número de personas de las que dispone el jugador, ya que sólo se puede realizar una acción por persona asociada.
- `(n-acciones ?j - jugador)`: Determina el número de acciones máximas que puede hacer un jugador ?j en cada turno. Este número solamente incrementará en caso de que el jugador amplíe su familia, encontrándose su rango en [1, 5]. Modela indirectamente el número de personas que posee un jugador, por lo que es el segundo parámetro del programa.
- `(n-recurso ?j - jugador ?r - recurso)`: Contabiliza el número de unidades del recurso ?r que posee actualmente el jugador ?j.

- `(n-celdas ?j - jugador ?c - celda)`: Contabiliza el número de celdas de tipo ?c que tiene el jugador ?j en el momento actual.
- `(n-pastos-completos ?j - jugador)`: Contabiliza el número de pastos vallados completos, es decir, aquellos pastos con dos animales dentro, de un jugador ?j en el momento actual de la partida. Esta función es indispensable para poder colocar animales en los distintos pastos vallados disponibles.
- `(n-campos-sembrados ?j - jugador)`: Indica el número de campos sembrados que posee el jugador ?j en el momento actual. No recogemos información del número de unidades de vegetal que se encuentran en cada campo porque no existe la fase de `RECOLECCIÓN` en esta simplificación del juego.
- `(n-vallas ?j - jugador)`: Esta función indica el número de vallas que ha construido el jugador ?j hasta la fecha. Es necesario contabilizar el número de vallas porque cada jugador posee un máximo de 15.
- `(jugado ?p - persona ?j - jugador)`: Indica el último turno en el que la persona ?p perteneciente al jugador ?j realizó su acción. Esta función impide que una misma persona realice más de una acción en un mismo turno.
- `(coste-adq-mayor ?a - adq_mayor ?r - construccion)`: Indica las unidades necesarias del recurso de construcción ?r para obtener la carta de adquisición mayor ?a. Aquellas adquisiciones mayores consideradas "dobles" tienen dos de estas funciones asociadas, ya que necesitan dos recursos diferentes.
- `(hornear-pan-1 ?a - adq_mayor)`: Representa las unidades de comida obtenidas utilizando una unidad de cereal con la adquisición mayor ?a.
- `(hornear-pan-2 ?a - adq_mayor)`: Representa las unidades de comida obtenidas a cambio de dos unidades de cereal con la adquisición mayor ?a. Es necesario hacer una distinción entre ambas formas de obtener comida horneando pan porque PDDL no permite tener tuplas formadas por dos cantidades numéricas (cereal que se pide, comida que se obtiene).
- `(penalty ?j)`: Métrica específica para cada jugador. Contabiliza la parte de `total-cost` que está asociada a un jugador ?j. Con esta función se puede determinar cuál de los dos jugadores lo ha hecho mejor en la partida.

### 2.2.4. Métrica

La métrica, tal y como pide el enunciado, la hemos llamado `total-cost`. **Para determinar los costes de cada una de las acciones** que pueden realizar los jugadores en nuestra versión de *Agricola*, **hemos seguido una serie de pautas**. Dichas pautas están planteadas de modo que aquellas acciones más costosas sean aquellas consideradas 'peores', y viceversa. Es importante recalcar que ninguna de las acciones consideradas 'internas' posee coste alguno, tal y como explicamos anteriormente.

- **El coste se ha asignado tomando como única referencia el impacto de cada una de las acciones en la puntuación final del jugador**. Aquellas acciones que no repercuten directamente en la puntuación final (`cambiar-jugador-inicial` y `hornear-pan`) poseen un coste intermedio.
- En *Agricola*, **existen dos formas de contabilizar los puntos de cada uno de los logros que puede conseguir el jugador**. Así, para el número de recursos y celdas conseguidos se utiliza un rango entre -1 y 4 puntos; mientras que para el resto aspectos puntuables (por ejemplo, número de personas asociadas al jugador y puntos conseguidos por adquisiciones

mayores) se utiliza una escala a medida. A la hora de determinar la importancia de las distintas acciones en la puntuación final, se han utilizado medias aritméticas que tienen en cuenta cada una de las distintas fuentes de puntos que proporciona la acción. Así, y al tener dos escalas diferenciadas, hemos aplicado dos procedimientos distintos:

  - **Aquellas acciones que permiten obtener recursos y/o celdas han sido puntuadas con la media del número de recursos y/o celdas necesarios para conseguir la puntuación máxima (4)**. Por ejemplo, en el caso de la acción `obtener-recurso-animal`; se necesitan 8 ovejas, 7 jabalíes y 6 vacas para conseguir la máxima puntuación en cada uno de esos recursos. Por tanto, la puntuación que recibiría esta acción sería 7.
  - **Para el resto de acciones, simplemente se ha hecho una media de los puntos que pueden llegar a producir**. Por ejemplo, en el caso de `construir-habitacion`, la puntuación es diferente dependiendo del material que estén construidas. Así, con las habitaciones de madera se pueden conseguir hasta 0 puntos, con las de adobe 5 y con las de piedra 10. Por tanto, la media resultante sería de 5 puntos.

- Tras calcular los puntos de cada acción, **estas se han ordenado de manera descendiente, quedando las acciones que ofrecen más posibilidades de ganar las primeras** (al menos según nuestro criterio teórico), **o lo que es lo mismo, aquellas con más puntos**. De este modo, asociamos los costes de `total-cost` con una distancia uno (y empezando por uno también) según el orden de esta lista. Aquellas acciones con los mismos puntos tienen el mismo coste. Por último, comentar que es interesante ver que `ampliar-familia` obtiene el menor coste; lo cual tiene sentido, ya que si un jugador tiene una persona más, puede realizar más acciones por turno y así obtener más puntos.

- Los **costes finales** son los siguientes:

  - 1 → Ampliar familia.
  - 2 → Reformar habitación.
  - 3 → Obtener recurso animal.
  - 4 → Obtener recurso no animal, sembrar campo, cambiar jugador inicial, hornear pan.
  - 5 → Arar campo, construir habitación.
  - 6 → Vallar.
  - 7 → Obtener adquisición mayor.

### 2.2.5. Acciones

#### 2.2.5.1. Acciones del jugador

La primera y única acción que podía realizar el jugador en la **versión 0** era la de `jugar`. Por tanto, **todas las acciones del jugador que encontramos en la versión final están basadas en ella**. De este modo, estas acciones contienen las mismas precondiciones y efectos que la original; añadiendo únicamente las características propias de la acción. Para no repetirnos, explicaremos cómo funcionaba la acción original para centrarnos después exclusivamente las características únicas del resto de acciones. Esta acción tenía las siguientes precondiciones y efectos:

- `(jugar ?j ?p)`:

  - **Precondiciones**: El jugador ?j tiene el turno en el momento actual, la persona ?p pertenece a dicho jugador, esta persona todavía no ha realizado ninguna acción en ese turno y además el jugador no ha alcanzado el número de acciones que tiene disponibles por turno.

- **Efectos**: Queda indicado que la persona ?p del jugador ?j ha realizado su acción en el turno actual (`jugado ?p ?j`). Además, se incrementa el número de `acciones-realizadas ?j` en una unidad. Así, el jugador realizará siempre el mismo número de acciones que personas posea, pudiendo realizar dichas acciones cualquiera de las cinco personas que se le asignan al principio, aunque nunca repitiendo a ninguna de ellas en un mismo turno.

En los efectos de todas las acciones mencionadas a continuación se incluirá el incremento de `total-cost` y de `penalty ?j`, las dos métricas que hemos manejado en nuestra solución. **La forma de coordinar los costes de ambas funciones**, aunque en un principio se pensó en sumar sólo los costes individuales de las acciones a `penalty` y luego hacer la suma de ambos `penalty` en la acción `terminar`; **se hizo sumando ambas funciones de forma simultánea en todas las acciones del jugador**, ya que la otra forma no funcionaba correctamente.

- `(obtener-recurso-no-animal ?j ?p ?r ?ca)`: La persona ?p, perteneciente al jugador ?j, recoge 5 recursos (no animales; es decir, vegetales, de construcción o comida) del tipo ?r utilizando la carta ?ca, que permite obtener recursos no animales.

  Las diferencias con la primera práctica son claras, y es que aquí el jugador siempre recoge una cantidad fija de recursos al prescindir de la fase de `REPOSICIÓN`. En la versión de CLIPS, el jugador recogía tantas unidades del recurso como se hubieran acumulado. Estos cambios que veremos en todas las acciones se deben a la simplificación de estrategias explicada en el apartado de *Decisiones de diseño*.

  - **Precondiciones**: La carta ?ca tiene asociado un recurso ?r. El recurso ?r no es de tipo animal y la carta ?ca no ha sido usada.
  - **Efectos**: Incrementamos en 5 el número de recursos ?r del jugador ?j. Además, incrementamos el valor de `total-cost` y de `penalty ?j` en 4 unidades y marcamos la carta ?ca como usada.

- `(obtener-recurso-animal ?j ?r ?ca ?pa)`: La persona ?p, perteneciente al jugador ?j, recoge 2 recursos ?r de tipo animal si tiene alguna celda ?pa de tipo pasto libre, utilizando la carta ?ca, que permite obtener recursos animales.

  Esta acción queda incluso más limitada que la anterior con respecto a su primera implementación, ya que aparte de que el número de recursos recogidos se mantiene fijo, no permite intercambiar animales por comida cuando el jugador ya no disponga de pastos vallados en los que se puedan colocar los nuevos animales.

  - **Precondiciones**: Existe una carta ?ca con un recurso ?r asociado, que no ha sido utilizada. Además, el jugador ?j dispone de más celdas ?pa de tipo pasto que de pastos completos, por lo que hay espacio para poner los nuevos animales en algún pasto vallado vacío.
  - **Efectos**: Completamos un pasto del jugador ?j, añadiendo 2 unidades de recurso ?r al jugador ?p (todos los pastos quedan completos con dos animales al componerse estos siempre de una celda y no permitir la construcción de establos). Incrementamos el valor de `total-cost` y de `penalty ?j` en 3 unidades y se marca la carta ?ca como usada.

- `(arar-campo ?j ?p ?ca ?c ?v)`: Una persona ?p, perteneciente al jugador ?j, realiza la acción de la carta ?ca, que permite arar un campo, siempre que disponga de celdas ?v que estén vacías. Esta acción no presenta ninguna diferencia con la implementada en la versión de CLIPS.

  - **Precondiciones**: Existe una carta ?ca que no ha sido utilizada todavía y el jugador ?j dispone de al menos una celda ?v vacía en su tablero de granja.

  - **Efectos**: Reducimos en uno las celdas ?v vacías del jugador ?j a la vez que incrementamos su número de celdas ?c de campo en uno (no puede haber una celda con dos tipos a la vez). Incrementamos el valor de `total-cost` y de `penalty ?j` en 5 unidades y se marca la carta ?ca como usada.

- `(sembrar-campo ?j ?p ?ca ?c ?ce)`: Con esta acción, la persona ?p del jugador ?j puede sembrar un campo ?c con vegetales del tipo ?ce utilizando la carta ?ca, que permite sembrar campos.

  No necesitamos dos acciones de este tipo (como sucedía en CLIPS), ya que luego no vamos a recolectar los vegetales que produzca el campo. En la primera práctica necesitábamos dos reglas porque dependiendo de si se sembraba cereales u hortalizas, el campo producía un número u otro de vegetales. Consecuentemente, la única información que guardaremos de estos campos será cuántos han sido sembrados, ya que al no simular la fase de `RECOLECCIÓN`, estos nunca volverán a estar vacíos (y no se puede sembrar campos que no estén vacíos). Sin embargo, la estrategia de sembrar tantos campos como los recursos del jugador permitieran queda descartada por tratarse de una acción 'recursiva'.

  - **Precondiciones**: Existe una carta ?ca que no ha sido utilizada todavía y el jugador ?j dispone de más celdas de campo ?c que de celdas de campo sembradas. Esto quiere decir que aún posee alguna celda de campo sin sembrar, es decir, vacía. Además, para poder sembrar, el jugador también debe disponer de al menos una unidad del vegetal ?ce.
  - **Efectos**: Se reduce en una unidad el vegetal ?ce y se aumenta el número de campos sembrados del jugador ?j. Incrementamos el valor de `total-cost` y de `penalty ?j` en 4 unidades y se marca la carta ?ca como usada.

- `(vallar ?j ?p ?m ?ca ?c ?pa)`: Una persona ?p perteneciente al jugador ?j utiliza una carta ?ca y 4 unidades de madera ?m para vallar una celda vacía ?c y crear un pasto ?pa.

  Esta acción valla un pasto por completo tras una ejecución, colocando siempre 4 vallas; ya que en esta simplificación los pastos siempre se componen de una celda. A diferencia de la versión que implementamos en CLIPS, allí se colocaban tantas vallas como el jugador pudiera, es decir, hasta que se quedara sin madera o llegara al límite de vallas; pudiendo vallar más de un pasto (de una o dos celdas) por completo. Esta estrategia utilizaba acciones 'recursivas', por lo que no fue implementada.

  - **Precondiciones**: El jugador ?j dispone de al menos 4 unidades de madera ?m, no ha alcanzado el máximo de 15 vallas y dispone de alguna celda vacía ?c. Además, no se ha utilizado la carta ?ca, que permite vallar.
  - **Efectos**: El jugador pierde 4 unidades de madera ?m, que se convierten en 4 vallas. También se resta una celda vacía ?v del contador del jugador para pasar a ser una celda de tipo pasto ?pa. Por último incrementamos el valor de `total-cost` y de `penalty ?j` en 6 unidades y se marca la carta ?ca como usada.

- `(construir-habitacion ?j ?p ?ca ?r ?ju ?v ?h)`: La persona ?p perteneciente al jugador ?j construye una habitación nueva usando la carta ?ca (que permite construir habitaciones) y el mismo material de construcción ?r que el resto de sus habitaciones, además juncos ?ju; creando así una celda del tipo ?h.

  A diferencia de cómo se planteó en el Sistema Basado en el Conocimiento, aquí sólo se le permite construir una única habitación al jugador; mientras que en la versión anterior construía tantas como su número de materiales de construcción y el máximo de habitaciones le permitiera.

- **Precondiciones**: El jugador ?j debe tener al menos 5 unidades de recurso ?r y 2 unidades de juncos ?ju. Dicho recurso ?r debe coincidir con el material que están construidas las otras habitaciones del jugador. Además, debe disponer de alguna celda vacía ?v y tener menos de 5 habitaciones ?h construidas previamente, dado que este es el número máximo de habitaciones que puede tener un jugador en el juego original. Por último, la carta ?ca no se debe haber utilizado con anterioridad.
- **Efectos**: Se marca como usada una celda vacía, efectuando este cambio al reducir el número de celdas vacías ?v en uno y aumentando en uno también el número de celdas tipo habitación ?h. Se deben reducir las reservas de recursos en 5 para el tipo ?r y en 2 para los juncos ?ju. Por último, incrementamos el valor de `total-cost` y de `penalty ?j` en 5 unidades y se marca la carta ?ca como usada .

- `(reformar-habitaciones ?j ?p ?ca ?r1 ?r2 ?ju ?h)`: La persona ?p perteneciente al jugador ?j reforma todas las habitaciones de las que dispone, construidas con el material ?r1, al material que le corresponde ?r2 según las transiciones definidas, usando la carta ?ca, que permite reformar habitaciones. Las posibles transiciones son madera-adobe y adobe-piedra.

  Esta acción mantiene la misma funcionalidad que se definió en la primera práctica, aunque ahora es posible conseguirlo sin acciones 'recursivas' al llevar únicamente la cuenta de habitaciones y no crear una instancia para cada una de ellas.

  - **Precondiciones**: El jugador ?j dispone de habitaciones construidas con el material ?r1, siendo ?r2 el siguiente recurso en la transición de reformas. Debe disponer además de 5 unidades del recurso ?r2 y 2 unidades de juncos ?ju por cada celda de tipo habitación ?h que posee. Finalmente, la carta ?ca no debe haber sido utilizada con anterioridad.
  - **Efectos**: Se restan los recursos ?r2 y ?ju utilizados para la construcción y se cambia el tipo de material de las habitaciones de ?r1 a ?r2. Por último, incrementamos el valor de `total-cost` y de `penalty ?j` en 2 unidades y se marca como usada la carta ?ca.

- `(ampliar-familia-con-habitacion ?j ?p ?ca ?h)`: Una persona ?p del jugador ?j crea una nueva persona en su familia siempre que disponga de habitaciones suficientes, utilizando la carta ?ca, que permite realizar esta acción específica. La funcionalidad de esta acción (junto con la otra que existe para ampliar la familia) es la misma que vimos en la primera práctica.

  - **Precondiciones**: El número de celdas tipo ?h es mayor que el número de personas ?p del jugador ?j. Con esto nos aseguramos que a cada persona le corresponde una habitación. También debemos asegurarnos que la carta ?ca no se haya utilizado.
  - **Efectos**: Aumentamos el número de acciones disponibles para el jugador en cada turno en uno. Finalmente, se incrementa el valor de `total-cost` y de `penalty ?j` en 1 unidad y se marca la carta ?ca como usada.

- `(ampliar-familia-sin-habitacion ?j ?p ?ca)`: Al igual que la acción anterior, una persona ?p del jugador ?j crea una nueva persona en su familia utilizando la carta ?ca, que permite realizar esta acción específica. La diferencia radica aquí en que no es necesario comprobar si existe una habitación libre para esa nueva persona.

  - **Precondiciones**: No son necesarias más precondiciones que las básicas que componen la acción `jugar` y que la carta ?ca no se haya utilizado todavía, ya que no es necesario tener habitaciones extra ni recursos para realizar la acción.
  - **Efectos**: Los efectos son los mismos que en la acción anterior: aumenta el número de acciones disponibles para el jugador por turno en uno, aumenta el valor de `total-cost` y de `penalty ?j` en 1 unidad y se marca la carta ?ca como usada.

- `(cambiar-jugador-inicial ?j ?p ?ca)`: Una persona ?p del jugador ?j utiliza la carta ?ca para poder ser el jugador inicial a partir del turno siguiente. Esta acción mantiene la misma funcionalidad que la versión de CLIPS.

  - **Precondiciones**: No existen más precondiciones que las que necesitaba `jugar` aparte de comprobar que ?ca no se había usado antes en la partida; al no requerir esta acción de ningún tipo de recurso ni de que el jugador que la ejecute no sea el jugador inicial en el momento actual.
  - **Efectos**: Cambia el jugador inicial al jugador ?j, marca la carta ?ca como usada y se incrementa el valor de `total-cost` y de `penalty ?j` en 4 unidades.

- `(obtener-adq-mayor-un-recurso ?j1 ?j2 ?p ?ca ?a ?r)`: Una persona ?p perteneciente al jugador ?j adquiere una adquisición mayor ?a utilizando la carta ?ca (que permite esta acción) y las unidades del recurso de construcción ?r necesarias. No podemos evitar hacer una distinción entre adquisiciones mayores que necesitan un recurso de aquellas que necesitan dos (como en CLIPS) porque la segunda cuenta con más precondiciones y efectos que la primera.

  - **Precondiciones**: La adquisición mayor ?a no pertenece a ninguno de los dos jugadores (?j1, ?j2), nos aseguramos también que la adquisición mayor no es doble (es decir, que necesita sólo un recurso) y además que el jugador ?j1 dispone de las unidades del recurso ?r suficientes como para adquirirla. Finalmente, comprobamos que la carta ?ca no ha sido utilizada anteriormente.
  - **Efectos**: Se resta el número de unidades del recurso ?r que cuesta la adquisición mayor ?a del número almacenado en la función `n-recurso`, y se le otorga al jugador ?j1 con `(adq-mayor ?j1 ?a)`. Por último, se marca la carta ?ca como usada y se incrementan el valor de `total-cost` y de `penalty ?j` en 7 unidades.

- `(obtener-adq-mayor-dos-recursos ?j1 ?j2 ?p ?ca ?a ?r1 ?r2)`: Como sucedía en la acción anterior, una persona ?p perteneciente al jugador ?j adquiere una adquisición mayor ?a utilizando la carta ?ca (que permite esta acción) y, en este caso, las unidades de los recursos de construcción ?r1 y ?r2 necesarias. Estas adquisiciones mayores (tanto las de esta acción como las de la anterior) permiten hornear pan.

  - **Precondiciones**: Al igual que con las adquisiciones mayores de un recurso, debemos comprobar que la adquisición mayor ?a no pertenece ni al jugador ?j1 ni al ?j2, (asegurándonos que estos dos jugadores son diferentes). También debemos comprobar que la carta ?ca no ha sido utilizada todavía. Además, en este caso necesitamos comprobar que el jugador ?j1 dispone de las unidades necesarias de os recursos ?r1 y ?r2 para obtener la adquisición mayor ?a.
  - **Efectos**: Los efectos son homólogos a los de obtención de adquisiciones mayores de un recurso, pero con la diferencia de que para este caso debemos restar al jugador el coste de recursos de dos tipos diferentes, en lugar de solamente uno. Así, la carta ?a pasará a ser propiedad del jugador ?j1 y la carta ?ca quedará marcada como usada.

- `(hornear-pan-un-cereal ?j ?p ?a ?co ?r ?ca)`: La persona ?p, perteneciente al jugador ?j, haciendo uso de la adquisición mayor ?a que posee y de la carta ?ca que permite hornear pan, cambia una unidad de cereal ?r por una unidad de comida ?co.

  La funcionalidad de las dos acciones que permiten hornear pan es la misma que la de la primera práctica. Sin embargo, debido a las limitaciones de PDDL, aquí hemos tenido que dividir la acción original en dos, por no poder utilizar predicados con valores numéricos para establecer la relación entre los cereales exigidos y la comida obtenida.

  - **Precondiciones**: El jugador ?j debe poseer la adquisición mayor ?a `(adq-mayor ?j ?a)` y tener al menos una unidad cereales ?r. La carta utilizada para realizar la acción, ?ca, tampoco debe haber sido utilizada.
  - **Efectos**: Se resta el número de cereales en una unidad, se aumenta el número de unidades de comida del jugador tanto como aporte la adquisición mayor ?a. Por último, se marca la carta ?ca como usada y se incrementa el valor de `total-cost` y de `penalty ?j` en 4 unidades.

- `(hornear-pan-dos-cereales ?j ?p ?a ?co ?r ?ca)`: Como vimos en la acción anterior, el jugador ?j debe poseer la adquisición mayor ?a `(adq-mayor ?j ?a)` y tener, esta vez, al menos dos unidades de cereales ?r. No todas las cartas de adquisición mayor permiten hacer este intercambio, pero sí aquel que exige únicamente un cereal.
  - **Precondiciones**: El jugador ?j debe poseer la adquisición mayor ?a `(adq-mayor ?j ?a)` y tener al menos dos unidades de cereales ?r. También debemos comprobar que la carta ?ca no ha sido utilizada todavía.
  - **Efectos**: Se resta el número de cereales en dos unidades, se aumenta el número de unidades de comida del jugador tanto como aporte la adquisición mayor ?a. Por último, se marca la carta ?ca como usada y se incrementa el valor de `total-cost` y de `penalty ?j` en 4 unidades.

#### 2.2.5.2. Acciones internas del juego

Las siguientes acciones implementadas son necesarias para recrear funcionalidades básicas del juego. Por tanto, no son acciones que suceden en consecuencia de la utilización de cartas, sino que son **acciones implícitas para que funcione correctamente el flujo el juego**. La funcionalidad de las mismas quedó ya definida en la **versión 0** de nuestra implementación y son las siguientes:

- `(cambiar-jugador ?j1 ?j2)`: Permite cambiar de jugador cuando aquel que tiene el turno ha realizado todas sus acciones.
  - **Precondiciones**: El jugador ?j1 es el jugador actual y ha realizado todas las acciones que le permite el número de personas que posee en la actualidad. Es decir, el valor de la función `(acciones-realizadas ?j1)` coincide con el número de acciones que ha realizado el jugador en ese turno, `(n-acciones ?j1)`.
  - **Efectos**: El jugador actual pasa a ser ?j2 y ?j1 deja de serlo.

- `(cambiar-turno ?j1 ?j2)`: Cambia el turno del juego al siguiente cuando todos los jugadores (?j1 y ?j2) han realizado todas sus acciones. Resetea el número de acciones realizadas a 0 de ambos jugadores para que puedan volver a jugar el turno siguiente.
  - **Precondiciones**: Comprueba las precondiciones de `(cambiar-jugador ?j1 ?j2)` para ambos jugadores y que ?j1 se trata del jugador inicial. Esto indica que ambos jugadores han completado todas sus acciones en el turno actual.
  - **Efectos**: Incrementa el valor de `(turno-actual)` en uno, establece a ?j1 como `(jugador-actual)` (el jugador inicial es quien comienza a jugar en cada turno) y resetea las funciones de `(acciones-realizadas)` de ambos jugadores a 0.

- `(terminar ?j1 ?j2)`: Termina el juego cuando los jugadores ?j1 y ?j2 realizan todas sus acciones en el último turno.
  - **Precondiciones**: El turno actual coincide con el último turno del juego y los jugadores han realizado todas las acciones posibles durante dicho turno (esta última comprobación es la misma que hacíamos en `cambiar-turno`).
  - **Efectos**: Termina el juego, creando el predicado `(juego-terminado)`.

# 3. Pruebas realizadas

Para comprobar el correcto funcionamiento de la implementación en PDDL del juego *Agricola*, se han llevado a cabo seis pruebas bajo los nombres: *problema-X.pddl*, siendo X números del 1 al 6. **Todas las pruebas se han ejecutado habilitando la optimización '-O'**, ya que queremos que el planificador utilice la métrica que hemos definido y no la longitud del plan (métrica por defecto).

Los **objetivos** de estas pruebas han sido principalmente dos, debido a que disponíamos de un mayor número de pruebas que en la práctica anterior:

- Primero, consideramos que **el principal problema que presenta PDDL en este dominio es su rendimiento y escalabilidad respecto a pruebas más ambiciosas**. Es por ello que hemos centrado las tres primeras pruebas en comprobar cómo respondía en cuestiones de tiempo y número de estados según se pedían metas más complejas.
- El otro aspecto interesante a tratar es que, **como no estamos tratando con resultados que dependen de la aleatoriedad**, como sucedía en CLIPS; **para comprender cómo se comporta nuestro sistema con la métrica que hemos definido, debemos restringirlo y así forzar a que siga otras estrategias**. Esta mentalidad es la que hemos priorizado en las tres últimas pruebas.

## 3.1. Resultados

### 3.1.1. Prueba 1

En la primera prueba, hemos realizado una partida de dos turnos entre dos jugadores, con dos personas disponibles por jugador. El resto de objetos, predicados y funciones diferentes al número de personas y turnos se ha establecido igual que encontraríamos al principio de una partida de *Agricola*. Los resultados obtenidos fueron los siguientes:

- **Tiempo total de ejecución**: 1,00 segundo.
- **Número de estados**: 8.503 estados.
- **Longitud del plan**: 14 pasos.
- **Funciones objetivo**:

  `(PENALTY_J1)`: 17,00. `(PENALTY_J2)`: 22,00. `(TOTAL-COST)`: 39,00.

- **Plan**:

  1. `ARAR-CAMPO J1 P1-1 ARAR-UN-CAMPO CAMPO-1 VACIA-1`
  2. `CAMBIAR-JUGADOR-INICIAL J1 P1-3 JUGADOR-INICIAL-Y-O-ADQUISICION-MENOR`
  3. `CAMBIAR-JUGADOR J1 J2`
  4. `ARAR-CAMPO J2 P2-1 ARAR-UN-CAMPO-Y-O-SEMBRAR CAMPO-2 VACIA-2`
  5. `AMPLIAR-FAMILIA-SIN-HABITACION J2 P2-2`
     `AMPLIAR-LA-FAMILIA-SIN-TENER-HABITACIONES-LIBRES`
  6. `OBTENER-RECURSO-NO-ANIMAL J2 P2-3 JUNCO-2 UN-JUNCO`
  7. `CAMBIAR-TURNO J1 J2`
  8. `OBTENER-RECURSO-NO-ANIMAL J1 P1-1 PIEDRA-1 UNA-PIEDRA-1`
  9. `OBTENER-RECURSO-NO-ANIMAL J1 P1-2 MADERA-1 TRES-MADERAS`
  10. `CAMBIAR-JUGADOR J1 J2`

11. OBTENER-RECURSO-NO-ANIMAL J2 P2-1 PIEDRA-2 UNA-PIEDRA-2
12. OBTENER-RECURSO-NO-ANIMAL J2 P2-2 CEREAL-2 COGER-UN-CEREAL
13. SEMBRAR-CAMPO J2 P2-3 SEMBRAR-Y-O-HORNEAR-PAN CAMPO-2 CEREAL-2
14. TERMINAR J2 J1

#### 3.1.2. Prueba 2

Al igual que en la prueba anterior, se ha simulado una partida de *Agricola* entre dos jugadores, con dos personas para cada jugador; pero esta vez incrementando el número de turnos a tres. El resto de parámetros del fichero de problema han quedado intactos con respecto a la prueba anterior. Esta prueba nos servirá para **saber cómo escala el planificador cuando tiene que realizar un mayor número de acciones**. Los resultados obtenidos fueron los siguientes:

- **Tiempo total de ejecución**: 3,57 segundos.
- **Número de estados**: 11.068 estados.
- **Longitud del plan**: 21 pasos.
- **Funciones objetivo**:

  (PENALTY_J1): 25,00. (PENALTY_J2): 38,00. (TOTAL-COST): 63,00.

- **Plan**:

  1. ARAR-CAMPO J1 P1-1 ARAR-UN-CAMPO CAMPO-1 VACIA-1
  2. CAMBIAR-JUGADOR-INICIAL J1 P1-3 JUGADOR-INICIAL-Y-O-ADQUISICION-MENOR
  3. CAMBIAR-JUGADOR J1 J2
  4. ARAR-CAMPO J2 P2-1 ARAR-UN-CAMPO-Y-O-SEMBRAR CAMPO-2 VACIA-2
  5. AMPLIAR-FAMILIA-SIN-HABITACION J2 P2-2
     AMPLIAR-LA-FAMILIA-SIN-TENER-HABITACIONES-LIBRES
  6. OBTENER-RECURSO-NO-ANIMAL J2 P2-3 JUNCO-2 UN-JUNCO
  7. CAMBIAR-TURNO J1 J2
  8. OBTENER-RECURSO-NO-ANIMAL J1 P1-1 PIEDRA-1 UNA-PIEDRA-1
  9. OBTENER-RECURSO-NO-ANIMAL J1 P1-3 PIEDRA-1 UNA-PIEDRA-2
  10. CAMBIAR-JUGADOR J1 J2
  11. OBTENER-RECURSO-NO-ANIMAL J2 P2-2 CEREAL-2 COGER-UN-CEREAL
  12. OBTENER-RECURSO-NO-ANIMAL J2 P2-3 ADOBE-2 UN-ADOBE
  13. OBTENER-ADQ-MAYOR-UN-RECURSO J2 J2 P2-1
      REFORMAR-Y-DESPUES-UNA-ADQUISICION-MAYOR-O-MENOR COCINA-2 ADOBE-2
  14. CAMBIAR-TURNO J1 J2
  15. OBTENER-RECURSO-NO-ANIMAL J1 P1-1 HORTALIZA-1 COGER-UNA-HORTALIZA
  16. SEMBRAR-CAMPO J1 P1-3 SEMBRAR-Y-O-HORNEAR-PAN CAMPO-1 HORTALIZA-1
  17. CAMBIAR-JUGADOR J1 J2
  18. OBTENER-RECURSO-NO-ANIMAL J2 P2-1 MADERA-2 TRES-MADERAS
  19. VALLAR J2 P2-2 MADERA-2 CONSTRUIR-VALLAS VACIA-2 PASTO-2
  20. OBTENER-RECURSO-ANIMAL J2 P2-3 OVEJA-2 UNA-OVEJA PASTO-2
  21. TERMINAR J2 J1

#### 3.1.3. Prueba 3

En esta prueba hemos simulado una partida de *Agricola* entre dos jugadores, con dos turnos y tres personas por cada jugador. El objetivo de la misma no es más que **saber si escala mejor aumentando el número de personas por jugador o el de turnos**. Los resultados obtenidos fueron los siguientes:

- **Tiempo total de ejecución**: 3,55 segundos.
- **Número de estados**: 24.705 estados.
- **Longitud del plan**: 18 pasos.
- **Funciones objetivo**:

  `(PENALTY_J1)`: 32,00. `(PENALTY_J2)`: 27,00. `(TOTAL-COST)`: 59,00.

- **Plan**:

  1. `ARAR-CAMPO J1 P1-1 ARAR-UN-CAMPO CAMPO-1 VACIA-1`
  2. `ARAR-CAMPO J1 P1-2 ARAR-UN-CAMPO-Y-O-SEMBRAR CAMPO-1 VACIA-1`
  3. `CAMBIAR-JUGADOR-INICIAL J1 P1-4 JUGADOR-INICIAL-Y-O-ADQUISICION-MENOR`
  4. `CAMBIAR-JUGADOR J1 J2`
  5. `OBTENER-RECURSO-NO-ANIMAL J2 P2-1 JUNCO-2 UN-JUNCO`
  6. `OBTENER-RECURSO-NO-ANIMAL J2 P2-2 CEREAL-2 COGER-UN-CEREAL`
  7. `AMPLIAR-FAMILIA-SIN-HABITACION J2 P2-3 AMPLIAR-LA-FAMILIA-SIN-TENER-HABITACIONES-LIBRES`
  8. `OBTENER-RECURSO-NO-ANIMAL J2 P2-4 HORTALIZA-2 COGER-UNA-HORTALIZA`
  9. `CAMBIAR-TURNO J1 J2`
  10. `OBTENER-RECURSO-NO-ANIMAL J1 P1-1 MADERA-1 TRES-MADERAS`
  11. `VALLAR J1 P1-2 MADERA-1 CONSTRUIR-VALLAS VACIA-1 PASTO-1`
  12. `OBTENER-RECURSO-ANIMAL J1 P1-4 VACA-1 UNA-VACA PASTO-1`
  13. `CAMBIAR-JUGADOR J1 J2`
  14. `OBTENER-RECURSO-NO-ANIMAL J2 P2-2 PIEDRA-2 UNA-PIEDRA-1`
  15. `OBTENER-RECURSO-NO-ANIMAL J2 P2-3 PIEDRA-2 UNA-PIEDRA-2`
  16. `OBTENER-RECURSO-NO-ANIMAL J2 P2-4 ADOBE-2 UN-ADOBE`
  17. `OBTENER-ADQ-MAYOR-UN-RECURSO J2 J2 P2-1 UNA-ADQUISICION-MAYOR-O-MENOR COCINA-1 ADOBE-2`
  18. `TERMINAR J2 J1`

#### 3.1.4. Prueba 4

A lo largo de las tres pruebas realizadas anteriormente, pudimos comprobar que **las acciones del tipo** `obtener-recurso` (especialmente las de `obtener-recurso-no-animal`) **eran las que más se repetían**; debido a que la mayoría de las cartas pertenecen a esta acción y a que, en el caso de `obtener-recurso-no-animal`, no requieren ningún tipo de precondición que el jugador no cumpla ya al iniciar el juego. Es por ello que **decidimos eliminar todas aquellas cartas que permitían obtener recursos no animales, a excepción de los recursos de construcción** (ya que son necesarios para realizar muchas de las otras acciones). Decidimos que el número de turnos fuera tres y el de personas por jugador dos; ya que tres personas y dos turnos resultó ser más pesado para el planificador (ver pruebas 2 y 3).

- **Tiempo total de ejecución**: 5,31 segundos.
- **Número de estados**: 61.000 estados.
- **Longitud del plan**: 18 pasos.
- **Funciones objetivo**:

  `(PENALTY_J1)`: 30,00. `(PENALTY_J2)`: 27,00. `(TOTAL-COST)`: 57,00.

- **Plan**:

  1. `OBTENER-RECURSO-NO-ANIMAL J1 P1-1 PIEDRA-1 UNA-PIEDRA-2`
  2. `OBTENER-RECURSO-NO-ANIMAL J1 P1-2 JUNCO-1 UN-JUNCO`
  3. `CAMBIAR-JUGADOR J1 J2`
  4. `ARAR-CAMPO J2 P2-1 ARAR-UN-CAMPO-Y-O-SEMBRAR CAMPO-2 VACIA-2`
  5. `CAMBIAR-JUGADOR-INICIAL J2 P2-2 JUGADOR-INICIAL-Y-O-ADQUISICION-MENOR`
  6. `CAMBIAR-TURNO J2 J1`
  7. `ARAR-CAMPO J2 P2-1 ARAR-UN-CAMPO CAMPO-2 VACIA-2`
  8. `OBTENER-RECURSO-NO-ANIMAL J2 P2-2 MADERA-2 TRES-MADERAS`
  9. `CAMBIAR-JUGADOR J2 J1`
  10. `OBTENER-RECURSO-NO-ANIMAL J1 P1-1 ADOBE-1 UN-ADOBE`
  11. `OBTENER-ADQ-MAYOR-DOS-RECURSOS J1 J2 P1-2 REFORMAR-Y-DESPUES-UNA-ADQUISICION-MAYOR-O-MENOR HORNO-PIEDRA PIEDRA-1 ADOBE-1`
  12. `CAMBIAR-TURNO J1 J2`
  13. `OBTENER-ADQ-MAYOR-DOS-RECURSOS J1 J2 P1-1 UNA-ADQUISICION-MAYOR-O-MENOR HORNO-ADOBE PIEDRA-1 ADOBE-1`
  14. `OBTENER-RECURSO-NO-ANIMAL J1 P1-2 PIEDRA-1 UNA-PIEDRA-1`
  15. `CAMBIAR-JUGADOR J1 J2`
  16. `VALLAR J2 P2-1 MADERA-2 CONSTRUIR-VALLAS VACIA-2 PASTO-2`
  17. `OBTENER-RECURSO-ANIMAL J2 P2-2 OVEJA-2 UNA-OVEJA PASTO-2`
  18. `TERMINAR J2 J1`

#### 3.1.5. Prueba 5

Para hacer esta prueba partimos de la **misma problemática que en la anterior**: el planificador utiliza demasiado las acciones del tipo `obtener-recurso`. Sin embargo, aquí hemos enfocado la solución desde otra perspectiva, y es que, **si los jugadores necesitan recursos para comenzar a hacer acciones más interesantes, les podemos proveer de recursos desde el principio**. Así, se le otorgó a cada jugador 10 unidades de cada recurso de construcción.

- **Tiempo total de ejecución**: 1,30 segundos.
- **Número de estados**: 2.552 estados.
- **Longitud del plan**: 18 pasos.
- **Funciones objetivo**:

  `(PENALTY_J1)`: 24,00. `(PENALTY_J2)`: 23,00. `(TOTAL-COST)`: 47,00.

- **Plan**:

  1. `REFORMAR-HABITACIONES J1 P1-1 REFORMAR-Y-DESPUES-UNA-ADQUISICION-MAYOR-O-MENOR MADERA-1 ADOBE-1 JUNCO-1 HABITACION-1`
  2. `VALLAR J1 P1-2 MADERA-1 CONSTRUIR-VALLAS VACIA-1 PASTO-1`
  3. `CAMBIAR-JUGADOR J1 J2`
  4. `REFORMAR-HABITACIONES J2 P2-1 REFORMAR-Y-DESPUES-CONSTRUIR-VALLAS MADERA-2 ADOBE-2 JUNCO-2 HABITACION-2`
  5. `ARAR-CAMPO J2 P2-2 ARAR-UN-CAMPO-Y-O-SEMBRAR CAMPO-2 VACIA-2`
  6. `CAMBIAR-TURNO J1 J2`
  7. `ARAR-CAMPO J1 P1-1 ARAR-UN-CAMPO CAMPO-1 VACIA-1`
  8. `OBTENER-RECURSO-ANIMAL J1 P1-2 VACA-1 UNA-VACA PASTO-1`
  9. `CAMBIAR-JUGADOR J1 J2`
  10. `OBTENER-RECURSO-NO-ANIMAL J2 P2-1 JUNCO-2 UN-JUNCO`
  11. `CAMBIAR-JUGADOR-INICIAL J2 P2-2 JUGADOR-INICIAL-Y-O-ADQUISICION-MENOR`
  12. `CAMBIAR-TURNO J1 J2`
  13. `OBTENER-RECURSO-NO-ANIMAL J1 P1-1 MADERA-1 TRES-MADERAS`
  14. `OBTENER-RECURSO-NO-ANIMAL J1 P1-2 ADOBE-1 UN-ADOBE`
  15. `CAMBIAR-JUGADOR J1 J2`
  16. `OBTENER-RECURSO-NO-ANIMAL J2 P2-1 PIEDRA-2 UNA-PIEDRA-1`
  17. `OBTENER-RECURSO-NO-ANIMAL J2 P2-2 PIEDRA-2 UNA-PIEDRA-2`
  18. `TERMINAR J2 J1`

#### 3.1.6. Prueba 6

Por último, hemos realizado una prueba **mezclando los conceptos aplicados en las dos pruebas anteriores**. Por un lado, **se han eliminado todas las cartas que permiten obtener recursos a excepción de las de recursos de construcción**. No pudimos eliminar todas las cartas de este tipo de acción porque sino los jugadores se quedaban sin acciones disponibles. Por otro lado, **le hemos otorgado a cada jugador 40 unidades de cada uno de los recursos que tienen disponibles**. Esto les va a permitir acceder a la mayoría de las acciones desde un primer momento (a excepción de las de `obtener-recurso`, por supuesto).

- **Tiempo total de ejecución**: 4,65 segundos.
- **Número de estados**: 6.405 estados.
- **Longitud del plan**: 18 pasos
- **Funciones objetivo**:

  `(PENALTY_J1)`: 26,00. `(PENALTY_J2)`: 30,00. `(TOTAL-COST)`: 56,00.

- **Plan**:

  1. `OBTENER-ADQ-MAYOR-DOS-RECURSOS J1 J2 P1-1 REFORMAR-Y-DESPUES-UNA-ADQUISICION-MAYOR-O-MENOR HORNO-ADOBE PIEDRA-1 ADOBE-1`
  2. `VALLAR J1 P1-2 MADERA-1 CONSTRUIR-VALLAS VACIA-1 PASTO-1`
  3. `CAMBIAR-JUGADOR J1 J2`
  4. `OBTENER-ADQ-MAYOR-DOS-RECURSOS J2 J2 P2-1 UNA-ADQUISICION-MAYOR-O-MENOR HORNO-PIEDRA PIEDRA-2 ADOBE-2`
  5. `REFORMAR-HABITACIONES J2 P2-2 REFORMAR-Y-DESPUES-CONSTRUIR-VALLAS MADERA-2 ADOBE-2 JUNCO-2 HABITACION-2`
  6. `CAMBIAR-TURNO J1 J2`
  7. `CONSTRUIR-HABITACION J1 P1-1 CONSTRUIR-HABITACIONES-Y-O-ESTABLOS MADERA-1 JUNCO-1 VACIA-1 HABITACION-1`
  8. `CAMBIAR-JUGADOR-INICIAL J1 P1-2 JUGADOR-INICIAL-Y-O-ADQUISICION-MENOR`
  9. `CAMBIAR-JUGADOR J1 J2`
  10. `ARAR-CAMPO J2 P2-1 ARAR-UN-CAMPO CAMPO-2 VACIA-2`
  11. `SEMBRAR-CAMPO J2 P2-2 ARAR-UN-CAMPO-Y-O-SEMBRAR CAMPO-2 HORTALIZA-2`
  12. `CAMBIAR-TURNO J1 J2`
  13. `OBTENER-RECURSO-NO-ANIMAL J1 P1-1 MADERA-1 TRES-MADERAS`
  14. `OBTENER-RECURSO-NO-ANIMAL J1 P1-2 ADOBE-1 UN-ADOBE`
  15. `CAMBIAR-JUGADOR J1 J2`
  16. `OBTENER-RECURSO-NO-ANIMAL J2 P2-1 PIEDRA-2 UNA-PIEDRA-1`
  17. `OBTENER-RECURSO-NO-ANIMAL J2 P2-2 PIEDRA-2 UNA-PIEDRA-2`
  18. `TERMINAR J2 J1`

## 3.2. Tabla-resumen

Mostramos aquí, para mayor comodidad del lector, una tabla-resumen con las características que definen a cada una de las pruebas y sus resultados obtenidos (a excepción del plan y los valores de las funciones `penalty`).

| ID | Resumen de la prueba | Estados | Tiempo | Longitud | `total-cost` |
|---|---|---|---|---|---|
| 1 | Partida normal de dos turnos y dos personas por jugador. | 8.503 | 1,00 s | 14 | 39 |
| 2 | Partida normal de tres turnos y dos personas por jugador. | 11.068 | 3,57 s | 21 | 63 |
| 3 | Partida normal de dos turnos y tres personas por jugador. | 24.705 | 3,55 s | 18 | 59 |
| 4 | No se pueden recoger recursos vegetales ni comida. (*) | 61.000 | 5,31 s | 18 | 57 |
| 5 | Se empieza con 10 recursos de cada tipo de construcción. (*) | 2.552 | 1,30 s | 18 | 47 |
| 6 | 40 recursos de cada tipo. Sólo se pueden recoger recursos de construcción. (*) | 6.405 | 4,65 s | 18 | 56 |

Cuadro 1: Tabla resumen de las pruebas realizadas

(*) Tres turnos y dos personas por jugador.

## 3.3. Análisis de los resultados obtenidos

Al realizar las distintas pruebas, hemos podido analizar las siguientes **observaciones** sobre el funcionamiento del planificador con nuestra métrica y sus limitaciones:

- **El planificador escala mucho peor al aumentar el número de personas por jugador que el número de turnos**. Si nos fijamos en las pruebas 2 y 3, en las que aumentamos el número de turnos y de personas a tres, respectivamente; en la primera se alcanzan los 11.000 estados, mientras que la segunda lo duplica, con más de 24.000 estados. Sin embargo, esto apenas repercute en el tiempo de ejecución (ambas pruebas se quedan alrededor de los 3 segundos). Por otro lado, y aunque no aparezca en las pruebas mostradas, en un principio las pruebas 4, 5 y 6 se iban a realizar con los parámetros de turnos y personas de la prueba 3, pero estas no terminaron después de 5 minutos de ejecución. Como podemos observar, si aplicamos la configuración de la prueba 2 a dichas pruebas, los resultados se obtienen sólo en unos segundos. ¿Por qué sucede esto? Su explicación es la ya presentada en la sección de *Decisiones de diseño*, en la que dijimos que aumentar el número de instancias de personas aumenta la combinatoria exponencialmente de cada una de las decisiones del planificador.

- **El planificador no es capaz de obtener resultados para una simulación de *Agricola* con más de tres turnos en el tiempo medio estimado de una partida del juego (30 minutos)**. Hicimos pruebas (no mostradas en esta memoria) en las que ampliábamos el número de turnos a realizar o el número de personas por jugador iniciales a cuatro. Sin embargo, el planificador no conseguía un plan tras más de una hora de espera si se incrementaba el número de turnos y en el caso de las personas, sólo lo conseguía tras reducir el número de turnos a uno.

- **El planificador realiza las mismas acciones y en el mismo orden siempre y cuando no cambiemos el número de personas por jugador**. Si nos fijamos en las pruebas 1 y 2, ambos jugadores están repitiendo las mismas acciones. Así, las únicas nuevas acciones que encontramos en la prueba 3 son las del último turno. Sin embargo, si nos fijamos en la prueba 3 esto vuelve a suceder, pero con matices. ¿Qué queremos decir con esto? Que en el primer turno ambos jugadores repiten las mismas acciones que en la prueba anterior, pero añadiendo una nueva (al tener una persona más). No obstante, a partir del segundo turno ambos planes siguen caminos diferentes. Esto tiene bastante sentido y nos lleva a la reflexión del primer punto de este apartado: *el número de instancias de personas aumenta la combinatoria exponencialmente de cada una de las decisiones del planificador.*

- **El planificador tiene mayor tendencia a utilizar primero aquellas acciones que apenas tienen precondiciones, a pesar de no ser las que tienen menor coste**. Como podemos observar en las pruebas 1, 2, 3 y 4; las primeras acciones siempre consisten de `arar-campo`, `cambiar-jugador-inicial` y `obtener-recurso-no-animal`. Es comprensible que esto suceda, ya que todas ellas se pueden realizar desde el inicio del juego. Sin embargo, acciones como `ampliar-familia` o `reformar` son escasas, a pesar de ser las que menos coste conllevan. Habría sido interesante que el planificador obtuviera los recursos necesarios para reformar sus habitaciones; pero esto no sucede hasta que les entregamos los recursos desde el primer momento (ver prueba 5). Por último, debemos recalcar que este patrón se ha repetido en la prueba 4, a pesar de que las únicas cartas para obtener recursos disponibles eran las de materiales de construcción. Así, podemos concluir que este comportamiento sólo se puede evitar si se les proporciona a los jugadores una cantidad de recursos abundantes (ver pruebas 5 y 6).
- **El planificador no siempre elige aquellas acciones con menor coste cuando dispone de gran cantidad de recursos para poder elegir más libremente desde el principio**. Esto se puede ver claramente al observar las pruebas 5 y 6: en la prueba 5 el planificador no duda en utilizar con prontitud las dos cartas que tiene disponibles para `reformar`; sin embargo, en la prueba 6, se decanta por empezar por la acción con mayor coste de todas: `obtener-adquisicion-mayor`. Es importante darse cuenta que la carta que utiliza esta última acción permite también `reformar`, por lo que se está perdiendo una oportunidad de minimizar el coste. Esto se refleja en los resultados de `total-cost`, donde consigue 47 en la prueba 5 y 56 en la 6, a pesar de realizar el mismo número de acciones. Esto sucede porque *Metric-FF* utiliza *best-first search* como algoritmo de búsqueda, el cual no es precisamente un algoritmo óptimo (que encuentra siempre la mejor solución).

# 4. Conclusiones técnicas

Hemos dividido la sección de conclusiones técnicas en dos partes fundamentales, cada una de ellas con un enfoque diferenciado. La primera de ellas se centra en la **técnica de planificación automática** y en los problemas que ha supuesto al desarrollar el problema propuesto. La segunda tiene como objetivo **hablar sobre nuestra implementación en particular**; mostrando sus fortalezas y debilidades, así como la fidelidad conseguida con el juego original.

## 4.1. Adecuación y limitaciones de *planning* para este problema

En este punto vamos a tratar por qué pensamos que **la planificación automática no es una buena técnica para simular un juego como *Agricola***; apoyándonos en los resultados que obtuvimos al realizar la misma tarea con Sistemas Basados en el Conocimiento.

- **El tiempo de ejecución es el factor limitante principal**. Como hemos visto a lo largo de la memoria, la complejidad y fidelidad al juego original han estado marcadas por el tiempo de ejecución de distintas pruebas que se han hecho a lo largo de la implementación. Esto se debe a la pésima escalabilidad del algoritmo de búsqueda que implementa *Metric-FF*, cuyas soluciones alcanzan tiempos de ejecución mayores al tiempo medio de una partida de *Agricola* en su versión familiar (alrededor de los 30 minutos) cuando el número de rondas o personas por jugador es muy 'elevado'. Este número 'elevado' de rondas y personas ha quedado establecido en cuatro. Sin embargo, debemos recordar que el juego original consta de 14 rondas (que se componen de otras fases de juego además de la fase implementada en esta versión, `JORNADA LABORAL`) y un máximo de 5 personas por jugador. Por tanto, la representatividad del juego queda mermada significativamente sólo por el tiempo de ejecución.

- **El sistema de representación de tipos está muy limitado con respecto a las ontologías que permite crear CLIPS**. A pesar de que se puede representar una taxonomía de tipos de estilo jerárquico similar a la que ofrece CLIPS, esta se queda corta al no permitir que dichos tipos dispongan de atributos. Las grandes perjudicadas de este inconveniente son las dos acciones de `hornear-pan` que hemos implementado. La funcionalidad de ambas acciones se podía condensar en una sola utilizando CLIPS, debido a que se podía indicar en los atributos de la propia acción cuántos cereales se necesitaban para obtener cuánta cantidad de comida. En PDDL, los sustitutos de los atributos son los predicados y las funciones. Sin embargo, dichos predicados no admiten valores numéricos en sus variables, lo que impide guardar información como la del intercambio de recursos que supone hornear pan en un solo predicado. Por otro lado, las funciones representan un valor numérico en sí mismas, pero sólo uno, por lo que es insuficiente.

- **El sistema de prioridades que permitía implementar CLIPS en sus reglas (inexistente en PDDL) simplificaba mucho la creación de estrategias**. En nuestro caso particular, nos basamos principalmente en las prioridades o `salience` para definir estrategias complejas en nuestro Sistema Basado en el Conocimiento, desarrollado en la práctica anterior. En PDDL no existe ningún tipo de prioridad para las reglas, así que lo único que se puede utilizar para modelar el flujo de acciones es la métrica y las precondiciones de las acciones (que también presenta CLIPS). Sin embargo, es mucho más sencillo definir el comportamiento deseado del sistema a partir de las precondiciones que de la métrica, ya que esta última puede producir resultados no deseados, como analizamos en el punto 4 del *Análisis de las pruebas realizadas*. No obstante, debemos admitir que la limitación del tiempo de ejecución no nos ha permitido definir este tipo de estrategias, por lo que no es un inconveniente que hayamos sufrido durante el desarrollo de la práctica.

- **PDDL no da cabida a la aleatoriedad, por lo que siempre generará el mismo plan dado el mismo fichero de problema y dominio**. La aleatoriedad era uno de los puntos fuertes que encontrábamos en CLIPS. A través de ella se podía simular la forma de obtener las cartas de acción en la fase de `PRINCIPIO DE RONDA` y además permitía situaciones muy diferentes entre ejecuciones, a pesar de que hubiera diferentes estrategias establecidas. Además, la aleatoriedad es un componente fundamental en cualquier juego de mesa o de cartas, ya que es lo que propicia que las partidas sean diferentes e interesantes para los jugadores. Así, la propuesta de PDDL solamente ofrece el 'mejor' plan de acciones para una situación determinada del juego, lo que no resulta interesante en un simulador; ya que un usuario puede aprender incluso más de jugadas no muy acertadas tras ver sus consecuencias.

- **La métrica que permite definir PDDL no depende del estado del juego**. Sin embargo, y teniendo en cuenta que la intención de la misma es guiar las acciones de ambos jugadores, no es muy realista que realizar una misma acción en diferentes estados de la partida sea igual de buena (o de mala). Si ya dijimos que la métrica podía llegar a proveer resultados no deseados, hacer que el coste de las acciones sea constante nos aleja mucho de que el plan obtenido sea el 'mejor' que ambos jugadores podían realizar dada la situación inicial que se les plantea.

- **No poder crear nuevas instancias de los diferentes objetos definidos en la taxonomía limita algunas acciones**. Este punto afecta en particular a la acción de `ampliar-familia`, ya que lo natural habría sido crear una nueva persona tras ampliar la familia. Sin embargo, estas 'personas' son objetos del dominio que deben ser inicializados únicamente en el fichero del problema. Esto nos llevó a inicializar en un principio más personas de las necesarias para que los jugadores tuvieran todas las personas que necesitaran cuando dispusieran de una nueva acción. Sin embargo, esto trajo problemas en los tiempos de ejecución, que resolvimos limitando el número de personas asociadas por jugador a una más con las que empezaba. Por tanto, si hubiéramos trasladado la forma de inicializar las personas a las celdas del tablero de

granja del jugador, habría sido inviable. Es por ello que en nuestra solución propuesta sólo utilizamos el número de celdas disponibles de cada tipo para realizar las distintas acciones.

- **La forma de representar las acciones en PDDL es bastante similar a las reglas de CLIPS**. Las acciones de PDDL cuentan con precondiciones y efectos, al igual que las reglas de CLIPS. Así, si se compara nuestra primera práctica con esta segunda tomando como referencia las acciones implementadas en la de PDDL, vemos que el conocimiento queda representado de forma homóloga en ambas, a pesar de utilizar formatos diferentes. Esto facilita enormemente el traspaso de conocimientos de una plataforma a otra, siempre que se tenga en cuenta que PDDL está mucho más limitada.

## 4.2. Calidad y representatividad de la solución aportada

En este último punto de las conclusiones **vamos a indicar las restricciones y el alcance y utilidad del programa implementado**. Por tanto, a parte de repasar algunas de las restricciones más significativas vistas ya en el apartado de *Decisiones de diseño*, hablaremos también de las virtudes que consideramos que tiene nuestra implementación.

- **La solución aportada representa únicamente la fase de `JORNADA LABORAL`, por lo que quedan excluidas fase tan importantes como la `COSECHA`.** La mayor limitación de nuestra implementación consiste en que sólo se ha representado una de las múltiples fases que presenta el juego, la `JORNADA LABORAL`. Esta es la fase más importante, ya que es donde el jugador realiza las acciones que definen su estrategia. Sin embargo, esta estrategia queda vacía al no contar con el resto de fases, especialmente la de la `COSECHA`, que tiene un gran impacto en la puntuación final del jugador.

- **La solución aportada no es fiel al juego original al permitir únicamente que cada carta se juegue una vez por partida.** A diferencia del juego real y de la previa implementación en CLIPS, la única manera compatible con tiempos de ejecución razonables consta de usar esta medida para reducir el número de instancias disponibles y reducir la complejidad del problema. Sin embargo, esta restricción tan fuerte no permitiría jugar todos los turnos del juego original; por lo que sólo tiene sentido en esta versión tan reducida.

- **Sólo se pueden llegar a simular un máximo de tres turnos**. Nuestra solución sólo permite obtener un plan de hasta tres turnos en un tiempo razonable. Si se definen cuatro turnos o incluso cuatro personas por jugador (incrementar el número de personas por jugador escala mucho peor que el número de turnos), la ejecución se estima que puede durar entre dos y tres horas. Llegados a este punto, le sería mucho más útil al usuario del sistema probar el juego físicamente, ya que sus partidas duran de media 30 minutos.

- **El juego sólo lleva a cabo estrategias básicas que no dependen del estado del juego**. Esta es una consecuencia natural de que la métrica sólo pueda definir sus costes de forma constante, tal y como explicamos en el apartado anterior de *Conclusiones técnicas*. Cualquier jugador, nuevo o experto, no jugaría de la misma manera en cualquier partida, por lo que esto resta realismo al simulador proporcionado.

- **Las acciones de que permiten `obtener-recursos` y `vallar` son las que se ven más afectadas con respecto a la versión de CLIPS**. Decimos esto porque eran unas de las acciones que funcionaban según el estado del tablero de granja del jugador. No obstante, esto es impensable en PDDL al poder incluir solamente estrategias que no dependan del estado del juego por la limitación del tiempo de ejecución. Así, por ejemplo, mientras que en CLIPS el jugador vallaba tantos pastos como su madera y el máximo de vallas le permitiera; en PDDL utiliza cuatro vallas siempre que disponga de cuatro unidades de madera y no sobrepase las 15 vallas.

- **La acción de `ampliar-familia` cumple las reglas del juego original, a excepción que permite trabajar a recién nacidos**. En el *Agricola* original, cuando un jugador amplía su familia, crea un recién nacido, que no puede trabajar hasta el turno siguiente (cuando ya es adulto). Sin embargo, añadir esta restricción habría supuesto crear más acciones que ralentizarían el tiempo de ejecución, por lo que nuestra versión permite trabajar a recién nacidos. Esto cambia totalmente la estrategia de los jugadores, que pueden tener una acción extra sólo por el hecho de realizar `ampliar-familia`.
- **Podemos concluir que, utilizando nuestra solución propuesta, el usuario no sería capaz de aprender la dinámica del juego** (ya que las reglas están muy distorsionadas en cuanto al *Agricola* original), **pero sí podría aprender las distintas acciones que ofrece** (al estar todas implementadas) **y al menos una forma de cómo se pueden jugar**. Con esto concluimos que nuestro programa no se puede considerar un simulador como tal, debido a su extrema simplificación. Por tanto, afirmamos rotundamente que PDDL (y la planificación automática) no es un buen sistema para representar este juego debido a las limitaciones tanto del lenguaje de programación como del algoritmo de búsqueda. Si se busca una forma de representar el funcionamiento de *Agricola* es mejor utilizar CLIPS (Sistemas Basados en el Conocimiento).

# 5. Comentarios personales

Con esta práctica hemos podido comprender las utilidades de los Sistemas de Planificación y contrastarlas con los Sistemas Basados en el conocimiento, pudiendo así hacer uso práctico y real de la teoría aprendida durante todo el curso. Si nos centramos en la conceptualización en PDDL, hemos aprendido a abstraer al mínimo el número de acciones necesarias y de instancias, incluso más que con CLIPS, ya que ha sido necesario tener siempre en cuenta el tiempo de ejecución.

Sin embargo, en relación a la implementación en PDDL, hemos encontrado algunas dificultades a la hora de utilizar este lenguaje de programación. Decimos esto por los errores de compilación, que en limitadas ocasiones indican la línea exacta donde se encuentra el problema. Esta dificultad nos ha hecho ser más meticulosas de lo normal con el código.

Finalmente, debemos decir que las pruebas utilizadas para comprobar que todo funcionaba correctamente han sido bastante pesadas de realizar; sobre todo aquellas que hicimos en un principio para comprobar el tiempo de ejecución de nuestra implementación. Muchas veces las mejoras que aplicábamos no surtían efecto y fue bastante complicado conseguir una versión que se adaptara a nuestras exigencias en funcionalidad y tiempo de ejecución.