Asignación de tareas a programadores usando PDDL

IA FIB @ UPC

Carlos Bergillos, Adrià Cabeza, Roger Vilaseca

Junio 3, 2019

Índice

1 Introducción

En esta práctica tendremos que resolver un problema de planificación usando *Metric FastForward* (*Metric-FF*). Éste es un planificador que nos permite resolver problemas de planificación definidos en lenguaje PDDL (*Planning Domain Definition Language*).

La modelización de los problemas PDDL se divide conceptualmente en dos partes:

- **Modelización del dominio:** Contiene las definiciones del "mundo" del problema. Definiremos aquí todo aquello que es invariante para cualquier instanciación del problema, por ejemplo los tipos, las acciones y los predicados disponibles.
- **Modelización del problema:** Contiene la información concreta del problema para el dominio dado. Pueden existir infinitos problemas para un mismo dominio. Por ejemplo, aquí definiremos los objetos que intervienen en el problema concreto, y definiremos el estado inicial y el estado objetivo.

Dada un modelizacion del dominio y una modelizacion del problema, el planificador se encargará de buscarnos un plan válido (si existiera), que contendrá una secuencia de acciones que nos llevarán del estado inicial al estado inicial.

Además, para ciertas extensiones de la práctica, no solo nos interesará buscar un plan válido cualquiera, sino que nos interesará encontrar un plan que optimice un criterio dado (una métrica).

2 Descripción del problema

En esta práctica nos encargaremos de planificar un proyecto de programación de gran envergadura. Deberemos repartir un conjunto de tareas a realizar entre los programadores disponibles.

En concreto, disponemos de T tareas de programación, cada una de ellas tiene un grado de dificultad asignado (de 1 a 3), y un tiempo estimado de realización estimado (en horas).

También disponemos de un conjunto P de programadores. Cada uno de ellos tiene asignado un grado de habilidad (de 1 a 3), que nos indica lo mucho o poco que éste está capacitado para resolver las tareas.

No queremos asignar a los programadores tareas mucho más difíciles de lo que para ellos están capacitados. En concreto, a un programador solo le podremos asignar tareas de como mucho una unidad más de dificultad de lo que nos indique su habilidad. En caso de que sea necesario asignar a un programador una tarea más difícil que su capacidad, la duración de la realización de la tarea se verá incrementada en 2 horas.

Además, todas las tareas deberán ser revisadas, para ello, hará falta una nueva tarea adicional. Esta nueva tarea de revisión será de la misma dificultad que la tarea original. Los programadores tienen también asociada una calidad (de 1 a 2). Si la tarea original es realizada por un programador de calidad 1, la nueva tarea de revisión durará 1 hora, si en cambio el programador era de calidad 2, la nueva tarea de revisión durará 2 horas. Para evitar una recursividad infinita, las nuevas tareas de revisión no requerirán a su vez de revisión, y su tiempo de realización no se verá penalizado por la habilidad del programador que la realiza.

3 Nivel básico

En esta primera versión del problema, solamente tendremos como objetivo que todas las tareas queden asignadas a algún programador, sin tener en cuenta las tareas de revisión asociadas, y sin intentar optimizar ningún criterio.

3.1 Dominio

3.1.1 Variables

Para la correcta resolución de este problema de planificación hemos visto conveniente trabajar con variables con tipo. En concreto, hemos necesitado 2 tipos, los cuáles hemos llamado **programador** y **tarea**.

- **programador:** Se utilizará para las variables que correspondan a cada programador del conjunto P.
- **tarea:** Se utilizará para las variables que correspondan a cada tarea del conjunto T.

Para las próximas extensiones ya no se requieren más cambios en las variables, esta será la configuración definitiva.

3.1.2 Funciones

- **habilidadProgramador:** Esta función nos servirá para definir y conocer la habilidad de un programador determinado. La habilidad de un programador se representa con un número entero 1, 2 o 3.
- **dificultadTarea:** Esta función nos servirá para definir y conocer la dificultad de una tarea determinada. La dificultad de una tarea se representa con un número entero 1, 2 o 3.

3.1.3 Predicados

- `(asignacion ?x - programador ?y - tarea)`

  Al programador `x` se le ha sido asignada la tarea `y`, por lo tanto, éste deberá ser el encargado de realizarla.

- `(tareaAsignada ?x - tarea)`

  Nos indica que la tarea `x` ha sido asignada a algún programador. Nos servirá para evitar asignar dos veces la misma tarea.

3.1.4 Acciones

- **asignar:** Esta acción nos sirve para asignar una tarea a un programador.

  Parámetros

  - `?p - programador`
  - `?t - tarea`

  Precondición

  - La tarea `t` no está asignada.
  - La habilidad del programador `p` debe ser más grande o igual que la dificultad de la tarea `p` más uno.

  Postcondición

  - La tarea `t` ha sido asignada al programador `p`.

3.2 Problema

3.2.1 Objetos

Para la modelización del problema tenemos que definir el conjunto de tareas y el conjunto de programadores de los que disponemos, declarando tantas variables (objetos) como requiramos.

3.2.2 Estado inicial

Para cada tarea:

- Definimos la dificultad de la tarea (1, 2 o 3).

Y para cada programador:

- Definimos la habilidad del programador (1, 2 o 3).

3.2.3 Estado final

Para el estado final, requerimos que todas las tareas estén asignadas:

```
(:goal (forall (?t - tarea) (tareaAsignada ?t))))
```

3.3 Juegos de prueba

3.3.1 Prueba estandard

Input

En este caso introduciremos un juego de prueba con algun programador que pueda realizar las tareas.

```
(define (problem test-01) (:domain task)
    (:objects p1 p2 p3 - programador
  t1 t2 t3 - tarea)
    (:init
        (= (habilidadProgramador p1) 1)
        (= (habilidadProgramador p2) 2)
        (= (habilidadProgramador p3) 1)

        (= (dificultadTarea t1) 3)
        (= (dificultadTarea t2) 3)
        (= (dificultadTarea t3) 3)
    )

    (:goal (forall (?t - tarea) (tareaAsignada ?t))))
```

Output

```
ff: found legal plan as follows
step    0: ASIGNAR P2 T3
        1: ASIGNAR P2 T2
        2: ASIGNAR P2 T1
```

Justifación

Como esperavamos se han asignado todas las tareas al programador 2 por que su habilidad le permite realizar las tareas de dificultad 3

3.3.2 Prueba imposible de realizar

Input

En este caso todos los programadores tendran habilidad 1 i todas la tareas dificultad 3. Esto debería probocar que todos ningún prgogramador pueda tener tareas asignadas.

```
(define (problem test-01) (:domain task)
    (:objects p1 p2 p3 - programador
  t1 t2 t3 - tarea)
    (:init
        (= (habilidadProgramador p1) 1)
        (= (habilidadProgramador p2) 1)
        (= (habilidadProgramador p3) 1)

        (= (dificultadTarea t1) 3)
        (= (dificultadTarea t2) 3)
        (= (dificultadTarea t3) 3)
    )

    (:goal (forall (?t - tarea) (tareaAsignada ?t))))
```

Output

```
best first search space empty! problem proven unsolvable.
```

Justificación

Cómo esperavamos no hemos encontrado ninguna solución porque los programadores no tienen suficiente habilidad para poder realizar las tareas..

4 Extensión 1

En esta extensión, vamos a añadir a nuestro programa la revisión de tareas. La revisión de tareas consiste en que cada vez que algún programador realice una tarea, esta deberá ser revisada por otro programador. La dificultad de la revisión será la misma que de la tarea y por lo tanto utilizaremos el mismo criterio para asignar un programador, para revisar que en la sección 3.

4.1 Dominio

4.1.1 Funciones

Para esta extensión vamos a utilizar las mismas funciones que en el apartado anterior.

4.1.2 Predicados

- `(asignacionTarea ?x - programador ?y - tarea)`

  Equivalente al predicado "asignacion" en la sección 3.1.3.

- `(asignacionRevision ?x - programador ?y - tarea)`

  Predicado utilizado para asignar a una tarea un programador para que este la revise.

- `(tareaAsignada ?x - tarea)`

  Igual que en la sección 3.1.3.

- `(tareaRevisada ?x - tarea)`

  Predicado para conocer si la tarea en questión ha sido revisada por algún programador.

4.1.3 Acciones

- **asignar:** Esta acción és equivalente a la acción "asignar" explicada en la sección 3.1.4. Donde el único cambio es el cambio de nombre del predicado de "asignacion" a "asignacionTarea".

- **revisar:** El objetivo de esta acción es a partir de las tareas ya asignadas, asignar una revisión a todas la tareas con un programador diferente del que ha tenido asignada la tarea, el cual debe tener habilidad suficientemente grande para poder revisarla.

  Parámetros

  - `?p - programador`
  - `?t - tarea`

Precondición

Para poder realizar esta acción se debe cumplir:

- La tarea `t` esta asignada.
- La tarea `t` no esta revisada
- La habilidad del programador `p` debe ser más grande o igual que la dificultad de la tarea `t` más uno.
- El programador `p` no puede ser el programador que ha realizado la tarea `t`.

Postcondición

- La tarea `t` ha sido revisada por el programador `p`.
- La tarea `t` está revisada.

4.2 Problema

4.2.1 Estado inicial

Para cada tarea:

- Definimos la dificultad de la tarea (1, 2 o 3).

Y para cada programador:

- Definimos la habilidad del programador (1, 2 o 3).

4.2.2 Estado final

El estado final remplazaremos la comprobación de de que todas las tareas estén asignadas, por la comprobación de que todas las tareas estén revisadas.

```
(forall (?t - tarea) (tareaRevisada ?t))
```

4.3 Juegos de prueba

4.3.1 Prueba estandard

Input

En este caso introduciremos un juego de prueba con suficientes programadores para poder realizar todas las tareas y revisarlas.

```
(define (problem test-01) (:domain task)
    (:objects p1 p2 - programador
  t1 t2 t3 t4 - tarea)
    (:init
        (= (habilidadProgramador p1) 2)
```

```
        (= (habilidadProgramador p2) 2)


        (= (dificultadTarea t1) 1)
        (= (dificultadTarea t2) 2)
        (= (dificultadTarea t3) 3)
        (= (dificultadTarea t4) 2)
    )

    (:goal (forall (?t - tarea) (tareaRevisada ?t))))
```

Output

```
ff: found legal plan as follows
step    0: ASIGNAR P1 T4
        1: REVISAR P2 T4
        2: ASIGNAR P1 T3
        3: REVISAR P2 T3
        4: ASIGNAR P1 T2
        5: REVISAR P2 T2
        6: ASIGNAR P1 T1
        7: REVISAR P2 T1
```

Justificación

Como podemos observar todos los programadores tienen un nivel suficiente para poder realizar cualquier tarea, por este motivo se asignan todas las tareas a un programador y todas las revisiones al otro.

4.3.2 Prueba imposible de realizar

Input

En este caso habrá un programador con habilidad 2 y otro con habilidad 1. También habrà una tarea de dificultad 3. Cómo las tareas para revisión necesitan un programador con suficiente habilidad no se encontrarà ninguna solución porque la tarea en question no tendrà ningún programador capaz de revisarla.

```
(define (problem test-01) (:domain task)
    (:objects p1 p2 - programador
  t1 t2 t3 t4 - tarea)
    (:init
        (= (habilidadProgramador p1) 1)
```

```
        (= (habilidadProgramador p2) 2)


        (= (dificultadTarea t1) 1)
        (= (dificultadTarea t2) 2)
        (= (dificultadTarea t3) 3)
        (= (dificultadTarea t4) 2)
    )

    (:goal (forall (?t - tarea) (tareaRevisada ?t))))
```

Output

```
weighted A* search space empty! problem proven unsolvable.
```

Justificación

Cómo esperavamos no hemos encontrado ninguna solución porque se asigna la tarea t3 al programador p2 y p1 no tiene suficiente habilidad para revisarla.

5 Extensión 2

En la siguiente extensión, partiendo de la anterior, tenemos como objetivo minimizar el tiempo total que se usa en resolver todas las tareas. El tiempo total se interpreta como la suma de las horas de las tareas, tanto las principales como las de revisión.
En este caso, minimizar significa otorgar las tareas dependiendo de la habilidad o calidad del programador para que las tareas o sus respectivas revisiones sean menos duraderas.

5.1 Dominio

5.1.1 Funciones

Puesto a que esta es la primera extensión donde observamos el concepto de tiempo, hemos tenido que añadir funciones además de las mencionadas anteriormente.

- **tiempoTarea**: Esta función nos servirá para definir y conocer el tiempo asociado a una tarea T.
- **tiempoTotal**: Esta función nos servirá para definir y conocer la suma de duración de todas las tareas. Es decir el tiempo total que representa el conjunto de tareas.

5.1.2 Predicados

En cuanto a lo que concierne a los predicados, utilizamos los mismos que en la extensión anterior.

5.1.3 Acciones

Además de las acciones utilizadas anteriormente hemos tenido que añadir **asignarDificil** y modificar **asignar** y **revisar** para añadir los conceptos de tiempo en nuestro sistema.

- **asignar**: Esta acción nos sirve para asignar una tarea a un programador que tiene un valor mayor o igual a la dificultadTarea. **Parámetros**
  - ?p - programador
  - ?t - tarea

  Precondición
  - La tarea t no está asignada
  - La habilidad del programador p es igual o mayor a la dificultad de la tarea.

  Postcondición
  - La tarea t ha sido asignada al programador p.

- Se ha incrementado el tiempo total en el tiempo que tarda en realizarse la tarea.

- **asignarDificil**: Esta acción es idéntica que la anterior pero con la diferencia de que esta sirve para asignar tareas a los programadores que tienen una habilidad igual a la dificultad de la tarea menos 1. En este caso se incrementa el tiempo total en el tiempo que tarda en realizarse la tarea con una penalización de 2 horas denido a su habilidad.

- **revisar**: Esta acción nos sirve para asignar una tarea de revisión a un programador. **Parámetros**

  - ?p - programador
  - ?t - tarea

  Precondición

  - La tarea t no está asignada
  - La tarea no debe estar revisada
  - La habilidad del programador p debe ser más grande o igual que la dificultad de la tarea p más uno.
  - El programador que revisa la tarea no puede ser el mismo que el que la ha realizado.

  Postcondición

  - Se ha asignado a un programador la tarea de revisión.
  - La tarea está revisada.
  - Se ha incrementado el tiempo total el tiempo que tarda en revisarse la tarea, el cual depende de la calidad del programador.

5.2 Problema

5.2.1 Estado inicial

Para cada programador:

- Definimos la habilidad del programador (1, 2 o 3).
- Definimos la calidad del programador (1 o 2).

Para cada tarea:

- Definimos cuanto tiempo tarda en hacerse.
- Definimos la dificultad de la tarea (1, 2 o 3).

Además inicializamos el valor de *tiempoTotal*.

5.2.2 Estado final

Para el estado final, de la misma manera que en la anterior extensión, requerimos que todas las tareas esten revisadas. No obstante, en este caso añadimos una métrica: **minimizar el valor de tiempoTotal**.

```
(:goal (forall (?t - tarea) (tareaRevisada ? t)))
(:metric minimize (tiempoTotal))
```

5.3 Juegos de prueba

5.3.1 Prueba que minimice según la habilidad del programador

Input

En este caso introduciremos un juego de prueba que muestre como nuestro sistema es capaz de minimizar el tiempo con las asignaciones realizadas según la habilidad del programador.

```
(define (problem test-01) (:domain task)
(:objects p0 p1 p2 - programador
t0 t1 - tarea)
(:init
(=  (habilidadProgramador p0) 1)
(=  (calidadProgramador p0) 1)

(=  (habilidadProgramador p1) 2)
(=  (calidadProgramador p1) 1)

(=  (habilidadProgramador p2) 5)
(=  (calidadProgramador p2) 1)

(=  (dificultadTarea t0)2)
(=  (tiempoTarea t0)6)

(=  (dificultadTarea t1)2)
(=  (tiempoTarea t1)2)

(= (tiempoTotal) 0 )
)
(:goal (forall (?t - tarea) (tareaRevisada ?t)))
 (:metric minimize (tiempoTotal))
)
```

Output

```
ff: found legal plan as follows
step    0: ASIGNAR P2 T1
        1: ASIGNAR P2 T0
        2: REVISAR P1 P2 T1
        3: REVISAR P1 P2 T0
plan cost: 10.000000
```

Justificación

En este caso podemos ver como se ha cumplido el objetivo de la extensión que era que se minimizara el tiempoTotal. Además podemos observar que en este caso se ha minimizado según la habilidad del programador: Se han asignado las dos tareas al programador 5 porque es el que tiene más habilidad y eso significa menos tiempo de revisión.

5.3.2 Prueba que optimize según la calidad del programador

Input

En este caso introduciremos un juego de prueba que muestre como nuestro sistema es capaz de minimizarel tiempo con las asignaciones realizadas según la calidad del programador.

```
(define (problem test-01) (:domain task)
(:objects p0 p1 p2 - programador
t0 t1 - tarea)
(:init
(=  (habilidadProgramador p0) 1)
(=  (calidadProgramador p0) 1)

(=  (habilidadProgramador p1) 1)
(=  (calidadProgramador p1) 2)

(=  (habilidadProgramador p2) 2)
(=  (calidadProgramador p2) 2)

(=  (dificultadTarea t0)1)
(=  (tiempoTarea t0)6)

(=  (dificultadTarea t1)1)
(=  (tiempoTarea t1)2)

(= (tiempoTotal) 0 )
)
```

```
(:goal (forall (?t - tarea) (tareaRevisada ?t)))
 (:metric minimize (tiempoTotal))
)
```

Output

```
ff: found legal plan as follows
step    0: ASIGNAR P0 T0
        1: REVISAR P1 P0 T0
        2: ASIGNAR P0 T1
        3: REVISAR P1 P0 T1
plan cost: 10.000000
```

Justificación

En este caso podemos ver como se ha cumplido el objetivo de la extensión que era que se minimizara el tiempoTotal. Además podemos observar que en este caso se ha minimizado según la calidad del programador: se le han asignado todas las tareas de realización al programador 0 ya que este tiene de calidad 1 y eso significa menos tiempo de revisión hacia sus otros compañeros.

6 Extensión 3

En la siguiente extensión, partiendo de la anterior, tenemos como objetivo limitar el número de tareas que podemos asignar a una persona a 2. Por consiguiente, aumentaremos el número de personas trabajando en paralelo.

6.1 Dominio

6.1.1 Funciones

En esta extensión tenemos que limitar el número de tareas por programador, de manera que necesitamos algun método para mantener la consistencia. En nuestro caso hemos decidido crear una función.

- **nTareasProgramador**:Esta función nos servirá para definir y conocer el número de tareas que esta realizando un programor p.

6.1.2 Predicados

Para esta extensión vamos a utilizar los mismos predicados que la anterior extensión.

6.1.3 Acciones

Para esta extensión vamos a utilizar las mismas acciones que en la anterior extensión.

6.2 Problema

6.2.1 Estado inicial

Para cada programador:

- Definimos la habilidad del programador (1, 2 o 3).
- Definimos la calidad del programador (1 o 2).

Para cada tarea:

- Definimos cuanto tiempo tarda en hacerse.
- Definimos la dificultad de la tarea (1, 2 o 3).

Además inicializamos los valores de *tiempoTotal* y el número de tareas de cada programador, *nTareasProgramador*.

6.2.2 Estado final

Para el estado final, de la misma manera que en la anterior extensión, requerimos que todas las tareas esten revisadas y además que se minimice el tiempo total. Además, añadimos también la restricción de que cada programador solo puede realizar como máximo 2 tareas.

```
(:goal (and (forall (?t - tarea) (tareaRevisada ?t))
(forall (?p - programador) (<= (nTareasProgramador ?p) 2))))

(:metric minimize (tiempoTotal))
```

6.3 Juegos de prueba

6.3.1 Prueba estándard

Input

En este caso introduciremos un juego de prueba estándard sin ninguna complicación para comprovar que se realiza correctamente el objetivo de la extensión.

```
(define (problem test-01) (:domain task)
(:objects p0 p1 p2 - programador
t0 t1 - tarea)
(:init
(=  (habilidadProgramador p0) 3)
(=  (calidadProgramador p0) 2)
(=  (nTareasProgramador p0) 0)

(=  (habilidadProgramador p1) 1)
(=  (calidadProgramador p1) 2)
(=  (nTareasProgramador p1) 0)

(=  (habilidadProgramador p2) 3)
(=  (calidadProgramador p2) 2)
(=  (nTareasProgramador p2) 0)

(=  (dificultadTarea t0)1)
(=  (tiempoTarea t0)4)

(=  (dificultadTarea t1)3)
(=  (tiempoTarea t1)6)

(= (tiempoTotal) 0 )
)
```

```
(:goal (and (forall (?t - tarea) (tareaRevisada ?t))
 (forall (?p - programador) (<= (nTareasProgramador ?p) 2))))
 (:metric minimize (tiempoTotal))
 )
```

Output

```
ff: found legal plan as follows
step    0: ASIGNAR P0 T1
        1: REVISAR P2 P0 T1
        2: ASIGNAR P1 T0
        3: REVISAR P2 P1 T0
plan cost: 14.000000
```

Justifación

En este caso podemos ver como se ha cumplido el objetivo de la extensión que era que cada programador tuviera como máximo 2 tareas.

6.3.2 Prueba imposible de realizar

Input

En este caso hemos creado un juego de prueba que resulte imposible de realizar ya que cada persona tendría que realizar más de 2 tareas.

```
(define (problem test-01) (:domain task)
(:objects p0 p1 p2 - programador
t0 t1 t2 t3 t4 - tarea)
(:init
(=  (habilidadProgramador p0) 2)
(=  (calidadProgramador p0) 1)
(=  (nTareasProgramador p0) 0)

(=  (habilidadProgramador p1) 1)
(=  (calidadProgramador p1) 1)
(=  (nTareasProgramador p1) 0)

(=  (habilidadProgramador p2) 1)
(=  (calidadProgramador p2) 2)
(=  (nTareasProgramador p2) 0)
```

```
(=  (dificultadTarea t0)2)
(=  (tiempoTarea t0)1)

(=  (dificultadTarea t1)1)
(=  (tiempoTarea t1)4)

(=  (dificultadTarea t2)1)
(=  (tiempoTarea t2)1)

(=  (dificultadTarea t3)3)
(=  (tiempoTarea t3)5)

(=  (dificultadTarea t4)1)
(=  (tiempoTarea t4)6)

(= (tiempoTotal) 0 )
)
(:goal (and (forall (?t - tarea) (tareaRevisada ?t))
 (forall (?p - programador) (<= (nTareasProgramador ?p) 2))))
 (:metric minimize (tiempoTotal))
 )
```

Output

```
Astar epsilon search space empty!
Bailing out.
```

Justificación

Tal y como esperábamos, al tener tantas tareas (8 si contamos su realización y su revisión pertinente) y tan pocos programadores (3) era imposible obtener una solución.

7 Extensión 4

En esta última extensión, partiendo de la anterior, tenemos como objetivo minimizar, además del número de horas, el número de programadores.

7.1 Dominio

7.1.1 Funciones

En esta extensión tenemos que conocer el número de trabajadores que tienen alguna tarea asignada por tal de poder minimizar este número. Para tener esta informacion hemos creado una función.

- **programadoresTotal**:Esta función nos servirá para contar el número total de programadores que tienen asignada alguna tares.

7.1.2 Predicados

Para esta extenison hemos añadido un nuevo predicado que nos permite conocer si ya hemos contabilizado al programador para el contador programadoresTotal.

- `(programadorContado ?x - programador)`

  El programador `x` ha sido contado para la función programadoresTotal.

7.1.3 Acciones

Además de las acciones utilizadas anteriormente hemos tenido que añadir **contar** y **nocontar**.

- **contar**: Esta acción nos sirve para incrementar la función programadoresTotal cuando un programador tiene alguna tarea o revisión asignada. **Parámetros**
  - ?p - programador

  Precondición
  - El programador `p` no ha sido contado.
  - Todas las tareas han sido revisadas.
  - El número de tareas del programador `p` és mayor a 0.

  Postcondición
  - El programador `p` ha sido contado.
  - Se ha incrementado en 1 el contador programadoresTotal en uno.
- **nocontar**: Esta acción nos sirve para no contar a los programadores sin tareas. **Parámetros**

– ?p - programador

Precondición

– El programador `p` no ha sido contado.
– Todas las tareas han sido revisadas.
– El número de tareas del programador `p` és igual a 0.

Postcondición

– El programador `p` ha sido contado.

7.2 Problema

7.2.1 Estado inicial

Para cada programador:

- Definimos la habilidad del programador (1, 2 o 3).
- Definimos la calidad del programador (1 o 2).

Para cada tarea:

- Definimos cuanto tiempo tarda en hacerse.
- Definimos la dificultad de la tarea (1, 2 o 3).

Además inicializamos los valores de *tiempoTotal*, *programadoresTotal* y el número de tareas de cada programador.

7.2.2 Estado final

En este caso cómo todas las tareas deben estar inicializadas para poder contar el número de programadores con tareas el estado final se cumplirà cuando se haya contado a todos los programadores y cuando todos los programadores realizen menos de 2 tareas. En esta ocasion minimizaremos el tiempo total y el número de programadores. Haciendo pruebas hemos encontrado que una onderación igual nos da buenos resultados, debido a que ambos contadores son parecidos.

```
(:goal (forall (?p - programador) (and (<= (nTareasProgramador ?p) 2)
(programadorContado ?p))))

(:metric minimize (+ (* (tiempoTotal) 0.5) (* (programadoresTotal) 0.5)))
```

7.3 Juegos de prueba

7.3.1 Prueba minimzar programador

Input

En este caso introduciremos un juego en que todos los programadores tienen la misma habilidad y calidad, y todas las tareas la misma dificultad. Queremos ver cómo no asigna ninguna tarea a uno de los programadores.

```
(define (problem test-01) (:domain task)
    (:objects p1 p2 p3 p4 - programador
  t1 t2 t3 - tarea)
    (:init
        (= (habilidadProgramador p1) 3)
        (= (habilidadProgramador p2) 3)
        (= (habilidadProgramador p3) 3)
        (= (habilidadProgramador p4) 3)

        (= (calidadProgramador p1) 1)
        (= (calidadProgramador p2) 1)
        (= (calidadProgramador p3) 1)
        (= (calidadProgramador p4) 1)

        (= (nTareasProgramador p1) 0)
        (= (nTareasProgramador p2) 0)
        (= (nTareasProgramador p3) 0)
        (= (nTareasProgramador p4) 0)

        (= (dificultadTarea t1) 3)
        (= (dificultadTarea t2) 3)
        (= (dificultadTarea t3) 3)

        (= (tiempoTarea t1) 3)
        (= (tiempoTarea t2) 3)
        (= (tiempoTarea t3) 3)

        (= (tiempoTotal) 0)
        (= (programadoresTotal) 0)
    )



    (:goal (forall (?p - programador) (and (<= (nTareasProgramador ?p) 2)
(programadorContado ?p))))
```

```
    (:metric minimize (+ (* (tiempoTotal) 0.5) (* (programadoresTotal) 0.5)))
    )
```

Output

```
ff: found legal plan as follows
step    0: ASIGNAR P1 T3
        1: ASIGNAR P2 T2
        2: ASIGNAR P3 T1
        3: REVISAR P1 P3 T1
        4: REVISAR P3 P2 T2
        5: REVISAR P2 P1 T3
        6: CONTAR P1
        7: CONTAR P2
        8: CONTAR P3
        9: NOCONTAR P4
plan cost: 15.000000
```

Justifación

Como esperavamos el programa no ha asignado ninguna tarea al programador p4.

7.3.2 Prueba minimizar programadores con calidad 2

Input

Finalmente haremos una prueba con programadores con habilidad 3 y calidad 1 y programadores con habilidad 3 y calidad 2. Esperamos que todos los programadores del segundo grupo no sean asignados o esten asignados a revisiones.

```
(define (problem test-01) (:domain task)
    (:objects p1 p2 p3 p4 - programador
  t1 t2 - tarea)
    (:init
        (= (habilidadProgramador p1) 3)
        (= (habilidadProgramador p2) 3)
        (= (habilidadProgramador p3) 3)
        (= (habilidadProgramador p4) 3)

        (= (calidadProgramador p1) 2)
        (= (calidadProgramador p2) 2)
        (= (calidadProgramador p3) 1)
```

```
        (= (calidadProgramador p4) 1)

        (= (nTareasProgramador p1) 0)
        (= (nTareasProgramador p2) 0)
        (= (nTareasProgramador p3) 0)
        (= (nTareasProgramador p4) 0)

        (= (dificultadTarea t1) 3)
        (= (dificultadTarea t2) 3)

        (= (tiempoTarea t1) 3)
        (= (tiempoTarea t2) 3)

        (= (tiempoTotal) 0)
        (= (programadoresTotal) 0)
    )



    (:goal (forall (?p - programador) (and (<= (nTareasProgramador ?p) 2)
(programadorContado ?p))))

    (:metric minimize (+ (* (tiempoTotal) 0.5) (* (programadoresTotal) 0.5)))
    )
```

Output

```
ff: found legal plan as follows
step    0: ASIGNAR P4 T2
        1: ASIGNAR P4 T1
        2: REVISAR P3 P4 T2
        3: REVISAR P3 P4 T1
        4: CONTAR P4
        5: CONTAR P3
        6: NOCONTAR P2
        7: NOCONTAR P1
plan cost: 10.000000
```

Justificación

Cómo esperavamos los programadores con calidad 2 no han sido asignados. Creando así la mayor minimización posible.

8 Conclusiones

Con la realización de esta práctica hemos podido ver y comprobar la potencia de los lenguajes de programación declarativos como PDDL. La gran ventaja de estos reside en el hecho de que nosotros no tenemos que preocuparnos de los algoritmos necesarios para encontrar la solución (eso será trabajo del planificador), solo debemos centrarnos en definir adecuadamente el dominio y el problema.

La realización de esta misma práctica en un lenguaje de programación imperativo hubiera resultado muchísimo más compleja y pesada. La introducción de cada una de las extensiones, con sus cambios de dominio y problema asociados, hubiera requerido profundos cambios en el código de los algoritmos de resolución y en las estructuras de datos utilizadas.

Separando y aislando el dominio y problema de los algoritmos de resolución conseguimos tener mucha más flexibilidad a la hora de definir el problema, y ganamos mucha facilidad para poder hacer cambios en este cuando queramos.

Anexos

A Generador de juegos de prueba

Generar a mano varios casos de prueba es una tarea ardua y monótona que puede ser fácilmente automatizable, así pues, para poder probar varios valores rápidamente, decidimos crear un generador de juegos de prueba. En nuestro caso hemos decidido utilizar el lenguaje de scripting **Python** para generar casos de prueba automáticamente.

Nuestro generador de pruebas admite tres parámetros distintos:

- *–programadores P*, donde P es el número de programadores que queremos
- *–tareas T*, donde T es el número de programadores que queremos
- *–extension E* donde E es la extensión que deseamos utilizar.

Finalmente se genera un documento llamado *fichero_generadoN.pddl* donde la N es la extensión que hemos seleccionado en los argumentos del programa.