UNIVERSIDAD
DE GRANADA

Técnicas de Sistemas Inteligentes
grado en ingeniería informática

PRÁCTICA 2

Satisfacción de Restricciones

Autor
Fernando Vallecillos Ruiz

DNI
77558520J

E-Mail
nandovallec@correo.ugr.es

Grupo de prácticas
TSI 1 Lunes 17:30-19:30

Rama
Computación y Sistemas Inteligentes

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2019-2020

1. Introducción

Se realiza un práctica compuesta por la creación de uno o más agentes para resolver un problema de *pathfinding* en el entorno de juego GVGAI (*BoulderDash*). El agente deberá calcular un camino desde su posición inicial a la salida con diferentes restricciones o sobrevivir durante un periodo determinado de tiempo. Todas las acciones tienen fuertes restricciones de tiempo. Se describen los objetivos en los diferentes niveles:

1.1. Ejercicio 1

En este ejercicio se debe averiguar el valor asociado a cada una de las letras usadas a partir de una ecuación. Se cuentan 10 letras diferentes. El enunciado afirma que cada letra tiene un valor entre 0 y 9 asociado. Además cada valor está asociado a una única letra. Este problema se podría generalizar a todo el alfabeto con diferentes valores o incluso repetidos. Se opta por solucionar solo este problema de forma simple. Se necesita un array para las diferentes letras y otro para los diferentes valores (dígitos). La posición en los arrays relaciona letra y valor. Con la restricción *alldifferent* aplicada a los valores se asegura que cada valor esté asociado a una única letra. La ecuación expone una suma de palabras formadas por las letras. Para trasladar esta restricción se transforman estas palabras a los números que representan. Tras esto, se aplica la restricción de igualdad impuesta por la ecuación lo que resulta en los valores de cada letra.

1.2. Ejercicio 2

En este ejercicio se debe averiguar un número X de 10 dígitos donde cada dígito representa el número de i en X, siendo i la posición del dígito en X (comenzando por 0). Para cumplir la restricción de 10 dígitos se utiliza un array de 10 posiciones. Cada celda del array tendrá un dígito del número final. Se itera sobre este array imponiendo la condición. Entonces el valor de $X[i]$ será igual al número de i en el array. Gracias a la función *count* se puede contar el número de i en X. Usamos esta función en cada celda para verificar la igualdad. Ya que este número está representado en un array, si se necesita usar el valor del número se debería convertir a entero.

1.3. Ejercicio 3

En este ejercicio se debe averiguar el horario de una clase con diferentes restricciones. Existen 6 profesores, cada uno con una franja de horario disponible. El aula está abierta desde las 9:00 hasta las 15:00. Cada profesor debe impartir una hora y solo un profesor puede impartir en una hora. Se trata de un problema simple de horarios. Primero se crea un array con el número de horas necesarias a cubrir, en este caso 6. Este array contiene el número asociado al profesor que imparte esa clase. Ya que todas las horas son contiguas y se permite establecer rangos de índices, podemos crear el array de horario con índices que representen las horas. Para representar los horarios de los profesores se decide un array multidimensional. Este array se compone de una fila por profesor y dos columnas. El índice de fila indica el número del profesor. Las columnas muestran el horario disponible del profesor para comenzar una clase. Es decir, el segundo valor de este horario especifica la última hora a la que el profesor puede comenzar una clase. De esta forma se facilita luego la aplicación de restricciones.

Dado que en este caso existen 6 profesores que deben impartir 1 hora en un total de 6 horas, se puede deducir que cada profesor imparte una hora diferente. Podemos agilizar la búsqueda aplicando esta restricción en el horario(*alldifferent*). Luego de esto, aplicamos las restricciones de horario para cada celda del horario. Se comprueba que para la hora asociada a la celda, el profesor este disponible según su horario. De esta forma se obtiene un array con índices concordes a las horas y con identificadores del profesor que debe impartir en dicha hora.

1.4. Ejercicio 4

En este ejercicio se debe de averiguar el horario de varios clases con diferentes restricciones. Existen cuatro aulas y cada una puede ser usada por un único profesor a la vez. Existen 3 asignaturas y 4 grupos en cada una. Cada grupo de cada asignatura debe ser impartido durante una hora. Cada grupo solo puede recibir una sola lección en una hora. Existen 4 profesores con diferentes horarios de disponibilidad y a que asignaturas y grupos imparte. Las cuatro aulas disponibles tienen un horario de 9:00 a 13:00.

Para representar la solución se usan dos matrices con tantas filas como horas disponibles y tantas columnas como aulas disponibles. Una de las matrices representa la asignatura mientras la otra representa el grupo. Juntas establecen que asignatura y grupo se imparte en cada clase en cada hora. También se crea otra matriz auxiliar con el mismo tamaño que establece que profesor hay en cada aula para cada hora. En estas matrices se guardan valores que identifican los verdaderos valores de las variables. Para la matriz de asignatura se añade un valor extra, el

cuál significa que es un descanso.

Primero se asegura que un mismo grupo no pueda estar en diferentes aulas a la misma hora iterando por filas usando *alldifferent*. Se ha hecho igual con el horario de profesores por la misma razón. Luego se comprueba que cada cada combinación de asignatura y grupo aparezca una sola vez en la matriz(ninguna repetición). Tras esto imponemos que profesores pueden enseñar que combinación de asignaturas/grupos. Dada la distribución de los horarios, se sabe que habrá un total de 4 descansos. A estos descansos le asignaremos un identificador de profesor (diferente para cada uno de los descansos) para que la restricción anterior de los profesores(solo puede estar en una clase a la misma hora) funcione correctamente. Por último, se aplican la restricción de los horarios de los profesores. Es decir, aseguramos que no haya conflictos con sus horas restringidas.

1.5. Ejercicio 5

En este ejercicio se vuelve a averiguar unos horarios de un aula con diferentes restricciones para cada día de la semana. Existen 9 asignaturas impartidas en bloques de una o dos horas. Cada asignatura es impartida por uno de los 4 profesores. Cada profesor solo puede impartir un bloque de alguna de sus asignaturas cada día. Salvo el profesor 4, quien puede impartir un bloque de diferentes asignaturas en un mismo día. Por último, la cuarta franja horaria será un descanso.

Para representar la solución se usan dos matrices con tantas filas como horas disponibles y tantas columnas como días. Una de las matrices representa la asignatura mientras la otra representa el profesor. Juntas establecen que asignatura y profesor imparte cada hora cada día de la semana. Para la resolución de este problema se ha creado un array con el nombre de las diferentes asignaturas. A este vector se le ha añadido el descanso como si fuese una asignatura. Además, se ha añadido asignaturas auxiliares. Una por cada una de aquellas asignaturas que se impartan en bloques de 2 horas. También se añade un último array que tendrá la frecuencia de aparición de cada una de las clases en el horario (teniendo en cuenta el número de horas a impartir semanales y el tamaño de los bloques).

Primero retomamos una idea del ejercicio anterior. Para la asignatura del recreo y las continuaciones de las anteriores asignaremos profesores imaginarios para poder cumplir restricciones mas adelante fácilmente. Las demás asignaturas se asignan con su profesor correspondiente. Además, establecemos que los recreos son siempre en la cuarta franja horaria. Luego imponemos que las las asignaturas de dos horas no pueden comenzar en la franja anterior al recreo o en la última franja. Esto no sería obligatorio pero ayuda en la siguiente restricción. Se impone que las asignaturas con bloques de 2 horas estén seguidas de las asignaturas 'extras' creadas para esto. Simplemente iteramos en las primeras 5 franjas horarias

imponiendo esto. Se podría iterar sobre todas las franjas horarias, lo que permitiría suprimir la restricción anterior. Sin embargo, al estar mirando en casillas anteriores/posteriores al índice a iterar se tendrían que crear índices adicionales lo cual añade problemas en la lectura del programa. Tras esto, se comprueba que la frecuencia de aparición de las asignaturas en el horario es acorde con la calculada. Se impone la condición de que en cada día solo se puede impartir un bloque de la misma asignatura (con *alldifferent*). Por último, se aplica la restricción sobre el número de bloques que cada profesor puede impartir en un día(todos menos el cuarto, solo pueden impartir uno). Esta restricción también es impuesta con *alldifferent* gracias a que hemos asignado diferentes identificadores al profesor 4 según su clase.

1.6. Ejercicio 6

En este ejercicio se necesita repartir una serie de características entre 5 vecinos. Aunque es un problema que parece largo la implementación es bastante sencilla. Cada vecino tiene un total de 6 características. Cada una de ellas es única entre los vecinos. En este problema se tiene una matriz con tantas filas como número de vecinos y tantas columnas como características. Cada vecino guarda un identificador del valor por cada una de las características presentes. Este identificador señala la posición del valor según la característica en los arrays auxiliares (guardan los posibles valores de cada característica).

Primero se aplica la restricción de valores únicos en cada característica. Luego se comienzan a aplicar las restricciones en el mismo orden que aparecen en el enunciado. La mayoría son bastante directas y fáciles de traducir. Lo único que cabe destacar es el uso de variables locales. Estas se usan para aplicar relaciones entre vecinos en diferentes restricciones de los apartados. Serán variables con nombradas con el valor de alguna de las características y guardarán el valor del identificador del vecino que la posee.

1.7. Ejercicio 7

En este ejercicio se necesita encontrar la solución óptima en tiempo para un reparto de tareas. Estas tareas están asociadas a una duración y a un número de tareas previas que deben de estar completas para comenzar esta. Para esto se crean primero los arrays para contener las características de las tareas. Uno contiene la descripción y otro la duración. Se añade una matriz con dos columnas. Cada fila de la matriz representa una restricción de predecesores. Es decir, la tarea en la primera columna es predecesora de la tarea en la segunda columna. Se añaden otros dos arrays que especifican el tiempo de comienzo y finalización de cada una

de las tareas. Se calcula el máximo tiempo que pueden tardar todas las tareas (secuencial) para restringir el rango de tiempo en estos últimos arrays.

Se establece entonces el fin de cada tarea como el comienzo de esta más la duración que tome llevarla a cabo. Luego se añade la restricción sobre los predecesores. El instante de finalización de la tarea predecesora debe ser menor al instante de comienzo de la tarea a la que precede. Se utiliza la función *maximum* por comodidad para guardar el valor de la tarea que ha terminado mas tarde. Esto describe el tiempo total de todas las tareas y es el valor que se necesita minimizar.

1.8. Ejercicio 8

Este ejercicio es una variación del ejercicio anterior. Todos los datos y restricciones anteriores se mantienen. Se introduce el concepto de límite de trabajadores y trabajadores necesarios para una tarea. Al comienzo se añade el número de trabajadores disponible y un nuevo array indicando cuantos trabajadores se necesitan para cada tarea.

Se podría pensar que se necesitan añadir varias restricciones en esta variación. Sin embargo se da uso de la función *cumulative*. Esta función nos permite describir las restricciones en el reparto de recursos acumulados. Esta función recibe cuatro parámetros. Los tiempos de inicio de cada tarea. Las duraciones para cada tarea. El número de recursos necesarios para cada tarea mientras se ejecuta. El límite de recursos en cualquier momento. En nuestro caso, los recursos a repartir son los trabajadores, de los cuales tenemos 3. Manteniendo todas las restricciones anteriores y añadiendo esta nueva, se obtiene la nueva solución a esta variación.

1.9. Ejercicio 9

Este ejercicio es otra variación de los ejercicios anteriores. Se trata del problema *Flexible Job Shop Problem* o FJSP. En este problema cada tarea se vuelve a ejecutar por un solo trabajador. Sin embargo, cada trabajador toma un tiempo diferente en realizar la tarea. Teniendo en cuenta estas restricciones, se tiene que minimizar el tiempo total consumido en la construcción. Se procede a describir las nuevas estructuras de datos que se necesitan añadir o modificar. El array creado para la duración de las tareas pasa a ser 2D. Cada una de las columnas describe el tiempo consumido por el trabajador para realizar cada una de las tareas. Ya que las tarea tienen duraciones diferentes, se calcula el rango de tiempos válidos como el mínimo entre las sumas de las duraciones de cada trabajador. Es decir, se supone que solo uno de los trabajadores realiza todas las tareas y se escoge el menor tiempo de estos. De esta forma aseguramos que el tiempo total no sobrepase este

y por tanto se descartan soluciones no óptimas más rápido. Se creará un nuevo array de duraciones. Este array contiene el valor de las duraciones de cada tarea según el trabajador que la vaya a realizar. Se añade por último una matriz con el mismo tamaño que la matriz de duraciones. Esta matriz contiene el tiempo de comienzo de cada tarea según el trabajador que vaya a realizar. Es decir, aunque existan 3 trabajadores con 9 tareas (27 casillas posibles) solo 9 de ellas contendrán algún valor. Esta matriz describe que tareas realiza cada trabajador y cuando las comienza. Se utiliza para poder aplicar las restricciones de forma fácil y eficiente.

El problema se ha afrontado mediante una transformación a un problema más simple. En los ejercicios anteriores se ha visto como este problema se podría resolver si se tuviese un la misma duración para las tareas independientemente del trabajador. Por ello, se ha creado el nuevo array de duraciones. Como se ha descrito, este array contendrá el valor de las duraciones de las tareas dependiendo del trabajador que las vaya a hacer. Se comienza explicando la restricción de que cada trabajador solo puede estar en máximo una tarea en cada franja de tiempo. Para esto se usa una nueva función *disjunctive*. Esta función recibe dos parámetros, un array de tiempos de inicio y un array de duraciones. Esta función comprueba que para cada tarea especificada por cada punto de inicio, no haya ninguna otra tarea que se superponga sobre ella. Es decir los rangos de tiempo (tiempo inicio + duración) no pueden solaparse. Se itera por cada uno de los trabajadores, gracias a la matriz de tiempos de comienzo se puede obtener fácilmente los tiempos de inicio de cada uno de los trabajadores. Con el trabajador y el número de tarea, es trivial crear el array con los tiempos necesarios para cada tarea. Estos serán los argumentos de la función anterior por cada uno de los trabajadores. De esta forma se verifica que cada trabajador solo pueda trabajar en una tarea al mismo tiempo.

La siguiente restricción va a permitir asegurar que cada tarea se ejecuta una única vez y obtener el array de tiempos en cada tarea. Se utiliza una nueva función *alternative*. Esta función recibe cuatro parámetros. Los dos primeros son dos valores discretos mientras que los dos últimos serán dos arrays. Esta función comprueba que los dos primeros argumentos existan 1 sola vez en los otros dos arrays. De esta forma, los dos primeros argumentos serán el tiempo de comienzo de una tarea y la duración por un trabajador determinado. Los otros arrays se forman con los posibles comienzos de la tarea para los diferentes trabajadores (en la matriz de tiempos de comienzo) y la duración asociada si la hiciese cada trabajador. Esto se repetirá por cada una de las tareas. Esto permite que se obtenga el tiempo de comienzo de cada tarea y la duración de esta según el trabajador que la vaya a realizar.

Con esto se termina la transformación. Ya se tiene el array con el comienzo de cada una de las tareas y la duración de estas según el trabajador que las vaya a realizar. Se vuelven a repetir las restricciones de los ejercicios anteriores: uso de *cumulative* para máximo de 1 trabajador en una tarea, calcular el final de

cada tarea y mantener la precedencia. Con esto se habría resuelto el problema. Sin embargo existe una última variación. El *solver* por defecto tarda demasiado tiempo. Gracias a que este problema es famoso se ha podido leer que algunos *solver* no son apropiados para este problema. Cambiando el *solver* de *Geocode* a *Chuffed* permite resolver este problema en 0.2 segundos.

1.10. Ejercicio 10

En este ejercicio se necesita encontrar la preferencia máxima en una mochila con capacidad limitada. Es decir, se necesita encontrar una serie de objetos con un peso menor que la capacidad de la mochila y que a su vez, maximicen la preferencia obtenida. Primero se establecen tres arrays para describir los diferentes objetos, sus pesos y preferencias asociadas. También se crea un array 'booleano' que representa si un objeto entra o no dentro de la mochila. Este contendrá un 0 si no entra y 1 si entra. Por supuesto, también necesitamos una variable que guarde el valor de la capacidad máxima de la mochila.

Se aplican entonces una restricción muy sencilla. Se asegura que la suma de los pesos de los objetos en la mochila sea menor o igual a la capacidad de esta. Por último, se maximiza el valor de la preferencia de los objetos dentro de la mochila. Gracias al vector booleano podemos iterar sobre todos los objetos y tener en cuenta solo aquellos que han entrado en la mochila.