PRÁCTICA INTELIGENCIA ARTIFICIAL 19-20

Jorge Ríos Marfil - 100405942
Enrique Benvenutto Navarro - 100405806

ÍNDICE

INTRODUCCIÓN

En el siguiente proyecto buscamos poner en práctica dos técnicas aprendidas en Inteligencia Artificial para resolver un problema de reparto de tareas. En el supuesto, la Universidad está buscando mejorar el mantenimiento del campus, y pide a sus alumnos implementar una solución inteligente que realice el reparto y gestión diarias de las tareas que hay que realizar.

El sistema debe calcular la ruta, el orden de las herramientas a utilizar, y las tareas que debe realizar un trabajador a lo largo del día, de la manera más óptima posible y teniendo en cuenta varios parámetros que se le pasan. Se decide que por cada técnica empleada para resolver el problema se deben hacer dos implementaciones, una básica y una avanzada, que difieren en la cantidad de parámetros y variables que tienen que tener en cuenta para calcular la solución.

Antes de explicar qué son estos parámetros y para qué se utilizan, conviene explicar la estructura general del problema. Conviene pensar que el problema está dividido en “Estructuras Organizativas”, que en el código se representan como clases dentro del directorio *objetos.* Como veremos más adelante, las principales son *Trabajador*, *Tarea* y *Herramienta.* Esto se hace para simular la realidad de la mejor manera posible, al igual que en la vida real, en el programa un trabajador tiene propiedades como sus habilidades, las herramientas tienen una cantidad y un número de disponibles, además de otros atributos como su peso, y las tareas se tienen que realizar en una determinada área. Todas tienen más propiedades que las mencionadas, que se explicarán más adelante.

Es necesario tener una forma de modelar el campus en el código. Para ello, el mapa del área del campus se divide en áreas como se muestra a continuación, tardando de media cinco minutos en moverse entre zonas colindantes. Estas áreas se representarán en el código en una clase de tipo “enum” llamada *Areas*, que se encuentra también en *objetos*.

Al problema se le pasarán una serie de datos en un archivo de formato JSON, que se parseará para poder trabajar con ellos en Java. Estos datos contendrán toda la información necesaria del problema. La información que se proporciona de las clases es la siguiente:

De cada tarea: El tipo de tarea, el área, y las unidades de esa tarea.
De cada trabajador: El nombre del trabajador y su habilidad cuantificada sobre 10 de podar, limpiar, y reparar.
De cada herramienta: El nombre de la herramienta, el trabajo que realiza, su peso, su "mejora", y su cantidad.

Una vez explicados los parámetros de cada clase, podemos definir qué distingue el caso básico y el avanzado de cada técnica. En el caso básico, se tiene en cuenta la habilidad de cada trabajador para calcular cuanto tiempo tarda en realizar las tareas. Por ejemplo, si tiene una habilidad de 5 en limpiar, decimos que realiza 5 tareas de ese tipo cada hora.

Sin embargo, en el caso avanzado, además de tener en cuenta las habilidades de los trabajadores, hay que computar el problema sabiendo que hay varios tipos de herramienta para cada trabajo, y que las herramientas tienen propiedades como su peso, la mejora, o la cantidad que hay disponible de cada una, que el desplazamiento de los trabajadores cuando llevan una herramienta se ve afectado por el peso de la misma, que la tarea de poda genera restos orgánicos y que por tanto hay que generar una tarea de limpieza por cada tarea de poda realizada, y que se debe realizar la cuenta de horas trabajadas por trabajador y mostrarla al final de la ejecución.

Es importante saber que tanto en el problema básico como en el avanzado, los trabajadores parten del almacén, donde se guardan todas las herramientas.

Las técnicas de Inteligencia Artificial que vamos a implementar para resolver estos problemas son Inferencia y Búsqueda.

Inferencia es un proceso por el cual se define lo que llamamos una "base de hechos" y un conjunto de reglas compuestas por unas precondiciones y unas acciones que se realizan cuando las precondiciones se cumplen. Dichas reglas se ejecutan sobre la base de hechos y la modifican, pudiendo eliminar, modificar o añadir elementos a dicha base. El problema se da por terminado una vez que no queden reglas que se puedan ejecutar.

En IA hay varios tipos de búsqueda. Las dos principales que hemos estudiado son Búsqueda no Informada y Búsqueda Heurística. La que aplicaremos en esta práctica será la Búsqueda Heurística. Esta consiste en encontrar la solución más óptima posible a un problema basándose en un coste y en una heurística, que juntos forman lo que llamamos una Función de Evaluación. Partiendo de un estado inicial que será nuestro nodo raíz, vamos generando nodos hijo con todas las acciones posibles de los trabajadores y todos los estados posibles del problema, siempre recorriendo y ampliando los nodos del árbol con menor Función de Evaluación, hasta llegar al estado del problema al que queremos llegar, que denominamos "nodo meta".

Hemos dividido el documento por secciones, intentando que coincida la jerarquía del documento con la del proyecto, y manteniendo el nombre de métodos, clases y carpetas. En cada sección explicamos en detalle su funcionamiento.

JSON

Como se ha explicado anteriormente, toda la información del problema se le pasa a Java en formato JSON a través del archivo *problema.json*.

```
{
  "tareas":{
    "tareaLimpiar1":{
      "tipo": "limpiar",
      "area": "R",
      "unidades": 6
    }
  },
  "trabajadores":{
    "trabajador1":{
      "nombre":  "Antonio",
      "podar":   5,
      "limpiar": 5,
      "reparar": 5
    }
  },
  "herramientas":{
    "herramienta1":{
      "nombre":   "Tijeras de podar",
      "trabajo":  "podar",
      "peso":     0,
      "mejora":   0,
      "cantidad": 4
    }
  }
}
```

Ejemplo de archivo JSON

Para que Java pueda trabajar con ello, hace falta parsear los datos. Esto se hace desde la clase *LectorJSON*. En ella se definen unas listas donde se van a almacenar los objetos que guarden la información del JSON. Se creará un objeto por cada trabajador, herramienta y tarea. El método que lee el JSON, crea los objetos y los introduce en las listas es *readJSON*. Cuando estábamos trabajando con él, detectamos un error que tras consultar con los profesores, modificamos. Se trata de un error en la lectura de las herramientas, en el que erróneamente se están guardando los datos del campo “mejora”, en la variable que almacena la cantidad de una herramienta:

```
Long cantidadAux = (Long) herramientaObject.get("mejora");
```

Como supuestamente estamos asignando el valor de la cantidad, deberíamos estar leyendo el campo “cantidad” como se muestra a continuación:

```
Long cantidadAux = (Long) herramientaObject.get("cantidad");
```

problema2.json

Para probar la eficacia y eficiencia de nuestras soluciones implementadas, hemos creado otro archivo JSON con más tareas a realizar. Las pruebas las hemos estado realizando tanto con este problema como con el original proporcionado.

OBJETOS

En el directorio *objetos* hemos introducido aquellas clases de Java que vamos a reutilizar múltiples veces tanto en Inferencia como en Búsqueda, en la parte básica y en la avanzada. Como explicamos anteriormente, en él se encuentran las clases *Trabajador*, *Tarea*, *Herramienta* y *Areas*, junto con *Información* y *Debugger*, cuya finalidad y funcionamiento se explicarán un poco más adelante.

Clase *Trabajador*

La clase *Trabajador* contiene todas las propiedades que forman un trabajador, como su nombre, sus habilidades de poda, limpieza y reparación, así como los minutos que lleva trabajados, las unidades de trabajo realizadas, el área en el que se encuentra, un array de sus habilidades ordenadas de mayor a menor, así como un arraylist que apunta a la lista de tareas principales, que creamos principalmente para tener acceso a la misma, y la herramienta que tiene asignada.

Constructor

En el constructor de *Trabajador* la única modificación realizada es la inicialización de las variables que no venían inicialmente con el código y que hemos añadido.

setHabilidades()

En este método lo único que hacemos es una simple comparación entre las distintas habilidades de un trabajador, para poder insertarlas ordenadas de mayor a menor en la matriz de habilidades.

getHerramientas(int hab)

Este método devuelve la utilidad de una herramienta cuando se le pasa su posición en la matriz de habilidades. Solo lo utilizamos en la parte básica de Inferencia.

setMinutosTrabajados(Areas Area, int Unidades, int Avanzado)

Este método calcula los minutos trabajados por el trabajador. Lo primero que hace es calcular cuánto se va a tardar en realizar las tareas de un tipo en un área, teniendo en cuenta la habilidad del trabajador, y la mejora de su herramienta si es caso avanzado. Lo calcula multiplicando las unidades por 60 y dividiendo el resultado entre la suma de la habilidad y la mejora de la herramienta. A los minutos trabajados totales le suma esto y el tiempo de desplazamiento, que computa con el método *getCoste.* También llama al método *setUnidadesTrabajadas*, que actualiza el total de unidades trabajadas y si es el caso avanzado, genera una tarea de limpieza en ese mismo área si la tarea es de poda.

setMinutosTrabajados(String Area, int Avanzado)

Este método calcula los minutos que se tardan en ir de donde esté el trabajador, al área que se pase como parámetro, y se los añade a los minutos trabajados totales.

isMin(Tarea Tarea)

El método *isMin()* tiene una tarea bastante importante. Lo usamos en Inferencia para saber si una tarea es la más cercana a la que podemos ir. Esto nos ayuda a optimizar la ruta que cogemos, siempre yendo a las áreas con tareas del tipo que puede hacer nuestra herramienta. En el, hacemos comprobaciones básicas como si quedan unidades de la tarea que pasamos como parámetro, si la tarea es del tipo que podemos hacer, o si la tarea no está siendo realizada por nadie. Si cumple esto, recorre todo el arraylist de tareas y si encuentra una más cercana que cumple todas las condiciones necesarias, devuelve false, si no, devuelve true.

isAvailable()

Este método comprueba que una tarea está disponible y/o que no está siendo realizada por otra persona, para ello comprueba que quedan unidades de esa tarea por realizar, que la herramienta puede hacer esa tarea, y que está disponible para ese trabajador.

setTareas(ArrayList<Tarea> Tareas)

Este método hace una cosa muy simple pero útil, iguala el arraylist de tareas del trabajador al puntero del arraylist de tareas principal.

setUnidadesTrabajadas(Areas Area, int Unidades, int Avanzado)

Se le llama desde *setMinutosTrabajados(Areas Area, int Unidades, int Avanzado)*. Sirve para añadir las unidades realizadas de una tarea al contador total de unidades realizadas. Además, si el problema es avanzado, y la tarea realizada es de tipo podar, genera una de limpieza por cada una de poda en el mismo área.

Clase *Tarea*

La clase *Tarea*, como la de *Trabajador*, almacena toda la información de la misma. Tiene una variable adicional que hemos añadido llamada “asignada” que nos permite saber si una tarea está siendo realizada por una persona o no, y por cual si lo está. A parte de eso, todos los otros campos son los proporcionados en el problema.

Constructor

Lo único que hemos modificado de este es que inicializamos la variable “asignada” a null.

getDisponible(String nombre)

Este método devuelve true si la tarea no está asignada, o si está asignada a la misma persona que se pasa por parámetro.

isValid(String Tipo)

Se le pasa un tipo de herramienta, y devuelve true o false en función de si quedan unidades disponibles, la tarea no está asignada y el tipo de herramienta es el mismo que el de la tarea, o no. Se utiliza en inferencia.

Clase *Herramienta*

La clase Herramienta es otra de la cual hemos modificado muy poco en términos de parámetros. Lo único que hemos hecho es añadir uno llamado “disponibles” que simplemente nos dice la cantidad de herramientas de un tipo disponibles. Cuando un trabajador coja una de ese tipo, este número decrecerá en una unidad.

Constructor

Lo único que añadimos al constructor proporcionado es que igualamos “disponibles” a “cantidad”.

cogerHerramienta()

Resta una unidad a las herramientas disponibles, simula que el trabajador ha cogido una herramienta del tipo.

dejarHerramienta()

Suma una unidad a las herramientas disponibles, simula que el trabajador ha dejado una herramienta del tipo.

Clase *Area*

Es una clase muy simple de tipo enum. La hemos creado para que solo haya determinadas áreas posibles, puesto que si hiciéramos que el nombre de las áreas fuese un String, podríamos tener problemas de trabajar con áreas inexistentes.

Clase *Informacion*

Esta clase la hemos creado para principalmente obtener el coste de desplazarse de una celda a otra. Se compone de una matriz de adyacencia con todos los costes de ir de un área a otro, ilustrada en la siguiente tabla:

	R	J3	A	C2	J2	C1	U	J1	B
R	0	5	10	10	15	15	20	20	25
J3	5	0	5	5	10	10	15	15	20
A	10	5	0	5	5	10	10	10	15
C2	10	5	5	0	5	5	10	10	15
J2	15	10	5	5	0	5	5	5	10
C1	15	10	10	5	5	0	10	5	10
U	20	15	10	10	5	10	0	5	5
J1	20	15	10	10	5	5	5	0	5
B	25	20	15	15	10	10	5	5	0

getCoste(Areas Origen, Areas Destino, double peso)

Este método calcula el coste de desplazamiento de un área origen a un área destino utilizando el método *getIndex(Areas Area)* expuesto más abajo para obtener los índices de ambos áreas, buscar en la matriz de adyacencia el coste de ir de uno a otro, y, en el caso avanzado, añadirle además el peso de la herramienta.

getIndex(Areas Area)

Sirve para devolver la posición en la matriz de una celda.

Clase *Debugger*

Esta clase es adicional ya que en ningún momento se nos pide que la implementemos, pero mientras desarrollábamos el problema de inferencia e intentábamos debuggear el código para encontrar nuestros fallos no éramos capaces por lo que se nos ocurrió desarrollar nuestro propio debugger para así poder ir viendo qué va ocurriendo con las tareas, los trabajadores y las herramientas. Mostrando cada tipo de mensaje informativo en un color diferente, y que la indentación fuese la misma en todas las líneas, para que así fuera lo más legible posible.

Constructor

El constructor recibe los tres arraylist del problema, el de trabajadores, el de las tareas y el de las herramientas, es muy importante que los array que se le pasen sea sobre los que se está trabajando y no una copia de los mismos ya que sino no se actualizan los datos que se van a mostrar.

printTrabajadores()

Al llamar a esta función imprime por pantalla la siguiente información correspondiente al estado de los trabajadores en color azul:

- Nombre del trabajador.
- Unidades trabajadas.
- Minutos trabajados.
- Herramienta actual.
- Área en el que se encuentra.

printHerramientas()

Al llamar a esta función imprime por pantalla la siguiente información correspondiente al estado de las herramientas en color verde:

- Nombre de la herramienta.
- Trabajo que puede desempeñar.
- Unidades totales.
- Unidades disponibles.
- Mejora de la herramienta.
- Peso de la herramienta.

printTareas(boolean Todas)

Esta función puede ser llamada de diferentes formas, si se le pasa solo un parámetro ha de ser un booleano con el que eliges si quieres que se muestren todas o solo de las que queden unidades restantes, y en el caso que se le pasen dos parámetros el primero sigue siendo el mismo, el segundo es un string para filtrar por tipo de tarea. La información se mostrará en color cian y la información que muestra es la siguiente:

- Tipo de la tarea.
- Unidades restantes.
- Nombre del trabajador al que ha sido asignada.

Conclusiones:

Gracias a este método fuimos capaces de localizar gran cantidad de errores en inferencia, y después lo reutilizamos en la parte de búsqueda, aunque hay algunos parámetros que muestra que no son utilizados en búsqueda como el parámetro de asignación de las tareas. Esperemos que el método os resulte útil para poder ver el correcto funcionamiento del problema.

INFERENCIA

Se nos pide implementar el problema propuesto en el enunciado en java haciendo uso de la librería JEOPS, usando su motor de inferencia para resolverlo.

Problema Básico

En el problema básico se nos pide implementar las siguientes consideraciones:

- Cuatro trabajadores con habilidades específicas para cada tipo de tarea. Las habilidades indican la cantidad de tareas que es capaz de desarrollar el trabajador de ese tipo en concreto
- Tres tipos de herramientas las cuales sirven para un tipo de tareas en concreto, y unidades ilimitadas.
- Tres tipos de tareas, repartidas en 9 áreas diferentes, con un coste de desplazamiento entre áreas de 5 minutos por salto de área, y con un número de unidades en concreto.

Proceso de desarrollo

1. ***prototipo.rules***: En esta primera fase de desarrollo implementamos que solo hubiese un trabajador y solo desarrollar un tipo de tareas, para eso creamos dos reglas, que eran:

    a. Trabajador en el almacén, la cual es el caso inicial, en el que se encuentra el trabajador en el almacén sin herramienta, cogen una herramienta del tipo poda y se desplaza a un área “aleatorio” para desarrollar esa tarea(en esta fase del desarrollo quita todas las unidades de golpe).

    b. Movimiento entre celdas, el trabajador se encuentra en un área que no es el almacén, con una herramienta y busca un área que tenga unidades de trabajo del tipo de la herramienta que lleva.

    Con esto conseguimos que realice todas las tareas que hay (*prototipo1y2.json*), pero no vuelve al almacén todavía.

2. ***prototipo2.rules***: Realiza el mismo tipo de tareas que en el prototipo pero con una diferencia y es que ahora lleva un recuento de los minutos trabajados.

3. ***prototipo3.rules***: En esta fase ya se tienen en cuenta más consideraciones, ahora ya están implementados todos los tipos de tarea que hay y todas las herramientas del problema básico, además ahora ya vuelve al almacén al terminar su jornada y deja la herramienta. Para ello se han implementado las siguientes reglas:

    a. Tres reglas como la regla a del prototipo pero con una modificación, y es que ahora en cada regla coge una herramienta dependiendo de la habilidad del

trabajador, de tal forma que el trabajador cogerá siempre primero las tareas de su habilidad máxima ya que las reglas se ejecutan por orden prioridad. Y ahora en esta regla no sale en busca de una tarea sino que se queda en el Almacén.

b. Trabajador con herramienta en el almace, esta regla hace que el trabajador busque una tarea disponible que se corresponda con la herramienta que tiene, además que se desplaza a la tarea más cercana gracias a la función *ismin().*

c. Trabajador con herramienta, esta regla es igual que la del prototipo1-b pero con la diferencia que ahora se desplaza a la tarea más cercana.

d. Trabajador con herramienta sin tarea, esta regla se ejecuta una vez el trabajador ha realizado todas las tareas del tipo de la herramienta que lleva. En ese caso se desplaza al almacen y deja la herramienta.

4. ***reglas.rules***: estas ya son las reglas definitivas para el problema básico, en estas ya tenemos en cuenta a todos los trabajadores, para ello hemos añadido una variable a las tareas que indica a qué trabajador está asignado, de esta forma nos aseguramos que dos trabajadores no siguen la misma ruta de trabajo, y que no hay dos trabajadores trabajando en la misma unidad de trabajo. Para ello se añade la precondición el las reglas de que la tarea esté disponible para ese trabajador, la asignación de tareas se realiza cuando se llama a la función *isMin().*

Problema Avanzado

Consideraciones

El problema avanzado es una ampliación del básico al que se le añaden los siguientes requisitos:

- Mejora de las herramientas a la hora de realizar las tareas.
- Limitación del número de herramientas disponibles.
- Peso de las herramientas el cual relentiza al trabajador en los movimientos entre áreas.
- Generar una unidad de limpieza en el área en el que se ha realizado una tarea de poda.
- Modelar las horas trabajadas por cada trabajador(como no leímos a tiempo lleva implementado desde el problema básico).

Proceso de desarrollo

A la hora de implementar lo anterior en el *reglas.rules* seguimos el mismo orden que aparece en las consideraciones del problemas, a continuación les explicamos las fases del desarrollo:

1. Para implementar los tipos de herramientas duplicamos las tres clases del prototipo3-a, de tal forma que el trabajador coge las herramientas siguiendo el orden de su habilidad maxima e intentando coger primero las que tienen una mejora. Para tener en cuenta la mejora en el tiempo empleado, modificamos la clase minutos trabajados para que tuviese en cuenta la mejora de la herramienta.

2. Para limitar el número de herramientas disponible, creamos dicha variable en el objeto herramienta, de tal forma que en las precondiciones de las seis primeras reglas (es decir en las que el trabajador coge herramienta) se tiene en cuenta que las unidades sean mayores que 0, y una vez cogida se resta una unidad a las disponibles. Y por último cuando se vuelve al almacén se suma una unidad en las disponibles por si la quisiese usar otro trabajador.

3. Para tener en cuenta el peso de las herramientas, se modificó la función *setMinutosTrabajados*, y la función *getCoste* para que tuviesen en cuenta el peso de la herramienta a la hora de calcular el tiempo de desplazamiento. Aunque se modificó la función no influye al problema básico ya que en el problema básico se restringe las herramientas que se usan a las básicas, de tal forma que el cálculo del peso como el de la mejora al realizarse con 0 no se ven afectados.

4. Para añadir la tarea de limpieza una vez se ha podado se ha modificado la función *setUnidadesTrabajadas* de tal forma que cuando se realiza una tarea de tipo poda se busca en el arraylist de tareas ese mismo área pero para las tareas de limpieza y

se añade tantas unidades como unidades de poda se hayan realizado. Para que esto no influya en el problema básico se ha añadido un nuevo parámetro a la función, el cual es un entero que puede tomar los valores de 0 o 1 para que la sume o no respectivamente, lo ideal hubiese sido usar un booleano pero JEOPS no permite pasar como parametro true o false.

Esto nos produjo un fallo y es que una vez realizadas todas las unidades de una tarea no restablecíamos a null la asignación de la tarea, de tal forma que si en un inicio el trabajador A limpiaba una zona, si posteriormente se generaba una tarea de limpieza por culpa de que se había limpiado, solamente podía el trabajador A limpiar esa zona, para solucionarlo se ha creado una nueva regla con máxima prioridad (la primera del *reglas.rules*) que comprueba si una tarea tiene cero unidades y está asignada la desasigna, de esta forma puede limpiar cualquier trabajador los restos de la poda.

Para hacer que los trabajadores trabajen de forma más pareja, hemos decidido que las unidades se reduzcan de forma unitaria, de tal forma que en cada ciclo de JEOPS cada trabajador trabaja las mismas unidades. También se ha invertido el orden de las dos últimas reglas de tal forma que JEOPS primero comprueba si algún trabajador puede volver al almacén y después si algún trabajador puede trabajar, de esta forma pretendemos conseguir que ningún trabajador se quede esperado en mitad del campus, sino que vuelva al almacén y salga a ayudar a sus compañeros, y así se produzcan menos sobrecargas de trabajo.

BÚSQUEDA

En el problema de búsqueda heurística se nos pide resolver el mismo problema que en la parte de inferencia, pero esta vez implementandolo todo sobre java y las clases que ya se nos daban, para ello hemos tenido que rellenar las funciones que se nos daban en blanco y diseñando una heurística para cada problema.

Problema básico

Para el problema básico a diferencia del de inferencia en este tenemos que calcular el dia de trabajo solamente para un trabajador, en concreto para Antonio. Aunque las tareas a realizar, es decir el JSON sobre el que se ejecuta el problema es el mismo.

A la hora de implementar el problema no hemos necesitado más funciones y clases que las que necesitamos crear para el problema de inferencia.

Heurística

La heurística que hemos implementado para este problema ha tenido tres fases y en cada una de ellas hemos ido modificando la anterior, aunque existe una cuarta ya que en las tres primeras el método *equals* estaba mal implementado y generaba nodos de más:

1. Nuestra primera heurística lo unico que tenia en cuenta era el número de unidades restantes que había, para ello recorremos el array de tareas e íbamos sumando a la variable heurística. Esta heurística nos generaba los siguientes resultados:
    a. 1005 minutos trabajados
    b. 339 nodos explorados
    c. 119 nodos adyacentes
    d. 458 nodos en total
2. La siguiente mejora que implementamos fue hacer una estimación del tiempo que nos llevaría realizar esas tareas, para ello al recorrer el arraylist teníamos en cuenta el tipo de tarea que era, y dependiendo de eso calculamos el tiempo de trabajo en función de la habilidad del trabajador Antonio. Con esto conseguimos mejorar los resultado a los siguientes:
    a. 1005 minutos trabajados
    b. 278 nodos explorados
    c. 116 nodos adyacentes
    d. 394 nodos en total

3. Lo siguiente que implementamos fue una penalización por alejarse del almacén, con esto queremos conseguir que el trabajador realice siempre las tareas más próximas, y no que se vaya a la más lejana del mapa, después a la más cercana y así todo el rato, ya que eso generaría que se perdiese mucho tiempo en desplazamiento. Para implementarlo hacemos uso de la función *getCoste* de la clase *Información,* a la que le pasamos la posición del trabajador y el almacén (el último parámetro es cero ya que es el peso de la herramienta), y eso se lo sumamos a la heurística, con eso conseguimos los siguientes resultados:

    a. 1005 minutos trabajados

    b. 211 nodos explorados

    c. 107 nodos adyacentes

    d. 318 nodos totales

4. Después nos dimos cuenta que estábamos cometiendo un fallo en la función *equals* a la hora de comparar las herramientas, cuando nos dimos cuenta de esto y lo corregimos, el tiempo no se vio afectado pero el número de nodos se redujo drásticamente pasando a tener los siguientes resultados:

    a. 1005 minutos trabajados

    b. 128 nodos explorados

    c. 53 nodos adyacentes

    d. 181 nodos en total

5. Aunque una vez corregido el error nos dimos cuenta que en el problema básico meter la restricción del punto tres lo único que producía es que se generarán más nodos, aunque el tiempo no se vio afectado, siendo el siguiente nuestro resultado final:

    a. 1005 minutos trabajados

    b. 131 nodos explorados

    c. 10 nodos adyacentes

    d. 141 nodos en total

Con todo esto sacamos como conclusión que aunque se modifique la heurística y se bajen el número de nodos en ningún caso se ha visto afectado el número de minutos trabajados lo cual nos hace suponer que la solución del problema que estamos hallando es la solución mínima del problema.

Clases

Clase *AStar*

addAdjacentNodes(Node currentNode)

Este método sería equivalente a nuestro reglas.rules de la parte de inferencia, ya que controla la expansión de nodos dependiendo de la situación del nodo actual, en esta expansión hay tres situaciones diferentes:

1. Encontrarse el trabajador en el almacén sin herramienta, en esta situación se buscan tareas disponibles en las que el tipo sea la habilidad máxima de cada trabajador, en caso de que no haya tareas disponibles de su habilidad maxima se buscarán de sus siguientes habilidades. Una vez que se comprueba que haya unidades restantes de ese tipo de tareas, se pasa a buscar una herramienta que sea compatible con ese tipo de tarea y su mejora sea nula (ya que nos encontramos en el problema básico), una vez encontrada la herramienta se genera un nuevo nodo en el que se establece la herramienta del trabajador. Una vez creado el nodo se comprueba si ese nodo ya ha sido explorado, si no lo ha sido se procede a calcular la heurística, guardar quien es su predecesor, y por último se establece a nulo su sucesor. Finalmente se añade en la lista de forma evaluada a la *openlist.*

   En caso de que el trabajador no encuentre ninguna tarea disponible para ninguna de sus tres habilidades no se generará ningún nodo adyacente.

2. Esta situación la hemos denominado movimiento entre celdas, si el trabajador se encuentra en una celda con herramienta buscará otra celda en la que haya todavia unidades restantes por trabajar y que sean del mismo tipo que la herramienta que lleva actualmente, una vez encontrada esa tarea se crea un nuevo nodo. En ese nodo se establece a cero el número de tareas disponibles, se mueve al trabajador al área donde se encuentre la nueva tarea y se calcula los minutos trabajados, en los que se incluye el tiempo desplazamiento y el tiempo de trabajo.

3. Esta última situación sólo se da en el caso en el que el trabajador no haya encontrado ninguna tarea para su tipo de herramienta en el caso dos, si esto ocurre el trabajador vuelve al almacén, deja la herramienta y calcula el tiempo de desplazamiento hasta el almacén.

Clase *Node*

Node(Node original)

Esta función es la encargada de generar una copia de nuestro nodo, la función se nos da medio hecha y nosotros la tenemos que complementar con las variables extras que tengan nuestras clases, en este caso solo hemos modificado las clases *Herramientas* y *Trabajador* en el caso de *Herramientas* solo tenemos que copiar la variable extra de herramientas disponibles. En el caso de trabajador son muchas más variables, que son: los minutos trabajados, las unidades trabajadas, el área y por último la herramienta que tiene(en este caso no podemos copiarlo tal cual ya que entonces seguiría referenciando al objeto herramienta del nodo padre, por lo que tenemos que buscar esa herramienta en nuestro nuevo objeto)

computeHeuristic(Node finalNode)

Esta función implementa la heurística mencionada anteriormente, para ello recorremos todas las tareas disponibles y dependiendo del tipo de tarea sumamos a la heurística que inicialmente es cero la estimación en minutos de lo que nos queda por trabajar en función de la habilidad del trabajador.

equals(Node other)

La función equals es la función encargada de comprobar si dos nodos son semejantes para ellos nosotros hemos decidido que dos nodos son iguales cuando se cumplen las siguientes igualdades de atributos:

1. El número de herramientas disponibles es igual.
2. Las tareas tienen el mismo número de unidades restantes.
3. Los trabajadores se encuentran en el mismo área.
4. Los trabajadores llevan la misma herramienta(inicialmente comprobábamos que fuese el mismo objeto herramienta lo cual hacía que se generan nodos extra, ahora solo comprobamos que se llamen igual las herramientas, ya que nos resulta indiferente si tiene la herramienta número 3 o 1, solo nos interesa que sirva para lo mismo).

printNodeData(int printDebug)

Este método se emplea para ver información de cada nodo, nosotros hemos visto relevante mostrar los siguientes datos (dependiendo del valor de *printDebug*), con el valor 1 se muestra:

- Heurística del nodo.
- Coste del nodo.
- Evaluación del nodo.
- Estado actual de los trabajadores.

Con el valor dos en *printDebug* muestra toda la información anterior más la siguiente:

- Estado actual de las herramientas.
- Estado actual de las tareas.

computeEvaluation()

Esta función es muy sencilla ya que simplemente establece el valor de la variable evaluación, el cual es la suma de coste más heurística. Esta función ha de ser llamada despues de establecer el coste de la función y de calcular la heurística.

Clase *MainClass*

El *MainClass* se nos daba casi completado, nosotros hemos tenido que realizar algunas modificaciones en el para que nuestro programa funcionase correctamente, las modificaciones son las siguientes:

1. Hemos cambiado los “import’s” ya que había varios de ellos que apuntaban a las clases del problema avanzado de búsqueda por lo que lo hemos modificado para que use las clases del básico,

2. Hemos modificado el nodo meta, el cual tiene las siguientes características:

    a. Todas las herramientas en el almacén.

    b. Todos los trabajadores en el almacén.

    c. Todas las tareas sin unidades.

    Tanto la característica ‘a’ como la ‘b’ son compartidas con el nodo inicial por lo que no hemos tenido que modificar nada, mientras que la tercera no, para ello hemos recorrido todas las tareas para asignar sus unidades a cero.

3. En el método *printFinalPath* hemos añadido dos características para imprimir métricas del problema, que son el número de nodos que componen el path final y el porcentaje de nodos que forman el path final frente al número total de nodos explorados.

4. El método *printMetrics* lo hemos rellenado para que muestre las siguientes métricas:

    a. Número de nodos explorados.

    b. Número de nodos por explorar.

    c. Número total de nodos generados.

    d. Tiempo trabajado por Antonio tanto en minutos como en horas.

Problema Avanzado

En el problema avanzado de búsqueda se nos pide que implementemos todas las consideración del problema avanzado de inferencia, y ahora ya sí, tenemos que hacer que funcione con todos lo trabajadores y no sólo con antonio.

Aunque puede parecer un gran cambio más allá de la heurística que es “diferente” el resto del código ha sufrido pequeñas variaciones o ninguna, por lo que solo vamos a explicar los métodos que hayan sufrido alguna variación respecto al problema básico

Heurística

Inicialmente empezamos con la heurística del problema inicial para probar si el resto del código funcionaba para cuatro trabajadores, y así fue, el codigo funciono aunque como era de esperar la heurística no lo hizo ya que como estaba hecha para que antonio trabajase lo minimo asi fue, trabajaron todos menos antonio que no trabajaba nada.

El primer cambio que hicimos fue adaptar lo mismo que teníamos solo para un trabajador ampliarlo para los cuatro haciendo uso de un bucle, funcionó aunque seguiamos en las mismas había un trabajador que no trabajaba nada (aunque esta vez no era antonio), en este caso sí que fue útil añadir la penalización por alejarse del almacén ya que sino, el número de nodos se disparaba incluso dependiendo del problema que le pasásemos podía llegar a no converger.

El siguiente cambio que hicimos para compensar la carga de trabajo entre los trabajadores fue añadir una función que añadía una penalización por el desfase entre trabajadores, lo que hacíamos era calcular la media y sumarle a la heurística el desfase entre la media y los minutos trabajados de cada trabajador(aplicando el valor absoluto), de esta forma conseguimos hacer que los cuatro trabajasen pero seguía estando descompensado por lo que decidimos ampliar la penalización multiplicandola por una constante y llegamos a la conclusión que multiplicada por cinco era lo óptimo para que generase el mínimo de nodos pero aun así siguiese equilibrado.

En este punto funcionaba aunque nos faltaba por implementar que generase una unidad de limpieza cada vez que podase, cuando lo implementamos nos dejo de converger el problema, hasta el punto de que se ejecutaba de forma infinita o con problemas más básicos nos llegó a generar casi 200000 nodos. Por lo que modificamos la heurística y en la estimación del tiempo cuando fuese una tarea de poda lo tratábamos como una de poda y una de limpieza, de esta forma paso a converger el problema y funcionar perfectamente

Y ya por último pensamos otra forma de intentar equilibrar aún más la carga de trabajo entre los trabajadores y se nos ocurrió intentar usar la desviación típica, aunque no funcionó, y probando nos dimos cuenta que haciendo uso de la varianza multiplicada por una constante conseguimos equilibrar aún más la carga de trabajo.

Clases

Clase *AStar*

addAdjacentNodes(Node currentNode)

Este método mantiene la misma estructura que en el problema básico con ciertas modificaciones las cuales son las siguientes:

1. Eliminar la restricción del bucle for que limitaba a que solo generase nodos con Antonio, de esta forma pasaban a trabajar todos los trabajadores.

2. Eliminar la restricción que hacía que las herramientas que pudiese el trabajador fuesen solo de mejora igual a cero, de esta forma pasan a trabajar con todas las herramientas disponibles.

3. Implementar el control de unidades de herramientas, para ello añadimos la restricción a la hora de coger una herramienta de que sus unidades disponibles sean mayor que cero, posteriormente una vez asignada la herramienta al trabajador le restamos una unidad a las herramientas disponibles. Y ya por último al volver al almacén a dejar la herramienta añadimos una unidad a la herramienta que tuviese el trabajador.

4. La última modificación realizada es tener en cuenta que para cada unidad de poda realizada hay que añadir una unidad de limpieza para ello usamos la función del trabajador de *setMinutosTrabajados* pero esta vez el último parámetro es 1 para que lo añada. Aunque para ello antes hemos tenido que establecer el array de tareas en el trabajador.

Clase *Node*

computeHeuristic(Node finalNode)

A la hora de calcular la heurística hemos tenido que realizar diversos cambios para que funcionase con la nueva heurística pensada, y los cambios son los siguientes:

1. Calcular la media que usaremos posteriormente para la varianza

2. Crear un bucle con los trabajadores para así hacer la estimación de tiempo restante de trabajo con cada uno de ellos, la penalización de alejamiento del almacén, y el sumatorio para la fórmula de varianza.

3. Por último calculamos la varianza y se la sumamos a la heurística.

Conclusiones

Creemos que nuestro diseño de búsqueda heurística avanzado está muy optimizado en el número de nodos que explora, ya que el 85% de los nodos que genera forman parte del path final lo cual creemos que es una métrica muy buena, e incluso en problemas grandes como el del JSON *problemas2.json* que es casi el doble de tareas sigue siendo igual de preciso a la hora de explorar los nodos, y aún así por lo que hemos comparado con otros compañeros el tiempo máximo trabajado sigue siendo bajo por lo que creemos que es una muy buena solución.

CONCLUSIÓN Y OBSERVACIONES

A continuación adjuntamos una tabla con los tiempos que nos dan que trabaja cada trabajador en cada problema:

	Inferencia Básico	Inferencia Avanzado	Búsqueda Básico	Búsqueda Avanzado
Antonio	285	222	1005	225
Diego	205	295	N/A	315
Bernardo	250	265	N/A	280
Christian	313	224	N/A	283

Unidades en minutos

Al principio, nos costó adaptarnos bastante a JEOPS. A parte de que no hay documentación y los errores son poco descriptivos, es bastante restrictivo comparado con Java, y las limitaciones te fuerzan a ser creativo con lo que haces. Además teníamos que implementar toda la lógica principal en el mismo, que añadía un nivel adicional de problemas.

La parte de búsqueda fue un poco más fluida, y aunque nos atascamos un poco en la parte avanzada conseguimos hacer que nos funcionase.

La práctica sin duda alguna, es compleja y extensa. En total le hemos dedicado dos meses aproximadamente, trabajando cuatro horas los fines de semana y a veces entre semana también. No obstante, es un buen uso de lo aprendido en clase, y definitivamente sirve para quitarle la capa de polvo a los conocimientos de Java que teníamos y que a algunos se les empezaban a oxidar por falta de uso. El trabajo colaborativo es esencial para la práctica, y utilizar un programa de control de versiones como Git indispensable, es lo que nos ha permitido trabajar en sincronía, y saber exactamente quién ha hecho qué y cuándo.

Aunque difícil, el proyecto ha servido para poner en práctica y cementar los conceptos de Inferencia y Búsqueda Heurística, y ver cómo se pueden aplicar en la vida real.