UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

ESEIAAT

TÈCNIQUES DE INTELIGÉNCIA ARTIFICIAL Y APLICACIONES PARA LA AUTOMATITZACIÓN

Planificador de tascas productivas con búsqueda heurística

Autores:

Iván Pérez Castilla

Marc Prat Llorens

Jaime Tarrasó Martínez

Tutor: Bernardo Morcego Seix

Índice

1 Introducción

El siguiente proyecto consiste en resolver un problema de planificación de tareas a través de un algoritmo basado en la búsqueda heurística.

1.1 Problema proceso productivo

El problema en cuestión trata sobre un proceso productivo que, a partir de una lista de comandas con fechas de entrega, fabrique en el tiempo delimitado todos los productos. Cada uno de estos productos estará formado por 4 componentes que, una vez reunidos, requieren de un tiempo para fabricarse con distribución normal. De estos 4 componentes, hay 3 que llegan a través de una línea de producción, y existen hasta 9 tipos de componentes diferentes. El cuarto componente es común para todos los productos y siempre está disponible.

Los componentes serán los siguientes: “A”, “B”, “C”, “D”, “E”, “F”, “G”, “H”, “I”. De esta manera, en este proceso se pueden producir piezas tipo “ACD”, “EFA” o “HHD”, entre otros.

Para la resolución del problema es importante saber también que en este proceso los componentes llegan en lotes de 4 unidades. Por lo tanto, en caso de necesitar un componente “A” te llegará “AAAA” a la línea de producción.

Se dispone de un almacén para 16 productos donde se guardarán los componentes utilizados en el proceso, tanto si forman parte de un producto como si son componentes extra que se obtienen al ser los lotes de 4 unidades.

Tal y como se ha comentado, las comandas se realizarán con fechas de entrega y contienen una relación de los productos que se requieren. Seguidamente se encuentra una figura con el esquema del proceso productivo.

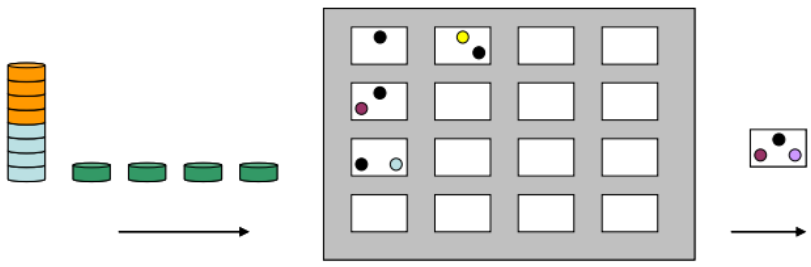

Ilustración 1Ejemplo proceso productivo

1.2 Objetivo

El objetivo de este proyecto es resolver el problema de manera óptima mediante un algoritmo de búsqueda heurística. En este caso, se utilizará el algoritmo A*.

Para conseguir un buen resultado, el algoritmo deberá retornar al usuario la secuencia óptima de entrega de los componentes a producir de manera que respete los tiempos de entrega de cada producto y no llegue a llenarse el almacén.

Se utilizará el programa Matlab para la programación del algoritmo.

1.3 Planificación del trabajo

La planificación del proyecto se muestra a continuación:

	Tiempo (Semanas)									
Descripción tareas	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Búsqueda de información y conocer el problema										
Planificar las tareas a realizar para llegar al objetivo										
Programación del algoritmo A*										
Creación del interfaz del usuario para hacer una comanda										
Creación del programa para crear listas de comandas que introduzca el usuario										
Programar una lógica heurística que tenga en cuenta las fechas de entrega de la comanda introducida por el usuario										
Programar una lógica heurística que tenga en cuenta la capacidad de almacén										
Programación del almacén simulado										
Relacionar los diferentes programas creados para que interactúen correctamente y encuentren la solución										
Realizar diferentes pruebas con diferentes datos del programa realizado										
Realizar cambios necesarios para mejorar el programa de búsqueda heurística										
Realización de la memoria										

2 Introducción de datos

En este apartado trataremos uno de los principales requisitos del sistema, la introducción de datos. Por concepto se ha decidido que la introducción de comandas se realizara una vez al día, de esta manera se introducirán todas a la vez y no habrá modificaciones durante la ejecución.

Para alimentar al programa necesitamos necesitaremos los siguientes parámetros:

- Número de componentes
- Número de pedidos
- Prioridad de los pedidos

Número de componentes:
En este apartado introduciremos la cantidad de componentes que queremos fabricar. El máximo posible en el problema será de 9 (A, B, C, D, E, F, G, H, I).

Número de pedidos:
Aquí introduciremos la cantidad de comandas que queremos introducir en el sistema para estudiar, por ejemplo 10.

Prioridad de los pedidos
Este apartado indica el orden de salida que daremos al producto, es decir, que prioridad queremos que tenga el producto Cuanto menos tiempo disponible para fabricarlo, más prioridad. Por ejemplo:

- Producto 1: 2 días
- Producto 2: 3 días
- Producto 3: 1 día
- Producto 4: 6 días

Una vez entrados todos los parámetros, ya se puede utilizar el algoritmo A* para la búsqueda del orden óptimo de pedido de los componentes de cada producto.

3 Algoritmo A*

El algoritmo A* es un algoritmo de búsqueda que puede ser empleado para el cálculo de caminos mínimos. Una de las principales cualidades de este algoritmo es su uso de una heurística que dará respuesta al problema a tratar. Cuando más acertada sea le heurística a usar, de mejor calidad será la solución encontrada.

La función heurística a desarrollar etiquetará los nodos que componen los posibles estados del problema. Sin embargo, estará compuesta a su vez por otras dos funciones. Una de ellas indicará la distancia actual desde el nodo origen hasta el nodo actual (g), y la otra expresará la distancia estimada desde el nodo hasta el nodo destino. Es decir, si se pretende encontrar el camino más corto desde el nodo

origen S (start), hasta el nodo destino G (final), un nodo intermedio tendría la siguiente función de evaluación f(n) como etiqueta:

f(n)= g(n) + h(n)

Donde:

- **g(n)** indica la distancia del camino desde el nodo origen S al n.
- **h(n)** expresa la distancia estimada desde el nodo n hasta el nodo destino G.

Por tanto, definir correctamente la función h será vital para una correcta implementación del algoritmo y una adecuada solución al problema en cuestión. Esta función h ha de estar estimada correctamente. Existen dos posibles casos a la hora de estimar esta función.

- ***Sobre-estimar h(n)***: la función h(n) implementada da estimaciones por encima de las que haría la función h(n) que verdaderamente modelaría este problema. El resultado obtenido utilizando el algoritmo A* no es "admisible" y no se puede garantizar que se encuentre el camino óptimo.

- ***Sub-estimar (o estimar correctamente) h(n)***: la función h(n) estima por debajo de los valores que estimaría la función h(n) que verdaderamente modelaría este problema. Por tanto, el resultado es "admisible" y encontrará un camino óptimo entre el nodo de inicio y de destino.

Con la función de evaluación f(n) definida, el algoritmo A* irá explorando los nodos en base a la función de evaluación. Estos se irán guardando en dos listas diferentes, Open y Closed, que guardarán los nodos aun por explorar y los explorados, respectivamente. Si la heurística está bien definida, ésta le marcará al algoritmo qué nodos se acercan al nodo destino, y encontrará más rápidamente el mejor camino hasta dicho nodo.

El algoritmo A* puede describirse de manera esquemática de la siguiente forma:

1. Establecer el nodo S como origen. Hacer f(S)=0, y f(n)=∞ para todos los nodos n diferentes del nodo S. Iniciar el conjunto O vacío.
2. Calcular el valor de f(S) y mover el nodo S al conjunto O.
3. Asegurar que el conjunto O no está vacío.
4. Seleccionar el nodo n del conjunto O que presente menor valor de la función f(n), eliminarlo del conjunto O y añadirlo al conjunto C.
5. Si el nodo n es el nodo destino finalizar algoritmo.
6. En caso contrario expandir el nodo n. Para cada nodo k expandido hacer:
    - 6.1. Calcular: f(n)=[g(n)+coste n hacia k] +h(n)
    - 6.2. Buscar si el nodo k está en el conjunto C.
        - 6.2.1. Si lo está, pasar al siguiente nodo expandido.

6.3. Buscar si el nodo k está en el conjunto O.
6.3.1. Si lo está pasar al siguiente nodo expandido.
6.3.2. Si no lo está, añadirlo al conjunto O.
7. Ordenar el conjunto O mediante f(n).
8. Si Q está vacío el algoritmo se termina. Si no está vacío, volver al paso 3.

3.1 Implementación

A continuación, se demostrará el algoritmo implementado utilizando código para ilustrar la metodología explicada previamente.

En Matlab utilizaremos una clase específica para tratar con los nodos, llamada "Node". Las variables internas que utiliza son:

```
classdef Node

    properties
        id
        component
        g
        h
        path
    end
```

Donde "id" será un identificador único de cada nodo, para poder localizarlo dentro de las listas; "component" sirve para asignarle un componente a dicho nodo; las variables "g" y "h" forman parte de f(n), y finalmente "path" indica qué camino ha tomado el nodo para llegar a su posición actual. Es decir, partiendo del nodo S, si el path de un nodo es: "SABCA", sabemos que el primer nodo era un componente "A", luego un "B", "C" y finalmente "A" de nuevo.

Por tanto, cada nodo de nuestro problema será un lote de 4 componentes que se pedirán. Eso significa que el camino obtenido a través del algoritmo nos indicará qué componentes, y en qué orden, han de pedirse.

Con la clase ya creada, el primer paso del algoritmo es inicializar los componentes a utilizar durante la función:

```
function [solution_node, previous_nodes, solution_path,
list_of_nodes] = Astar(initialH, components, lista_com)

id = 0;
```

```
goalN = Node;
goalN.id = id;
goalN.component = "G";
goalN.g = 10;
goalN.h = 0;
goalN.path = "G";

id = id + 1;

startN = Node;
startN.id = id;
startN.component = "S";
startN.g = 0;
startN.h = initialH;
startN.path = "S";

openList(1) = startN;
```

En primera instancia se crean los nodos de inicio (S) y de destino (G). El nodo de inicio será un nodo "vacío" que no tiene más peso en la algoritmia que el de servir como primer nodo. La heurística en este problema está definida de tal manera que el nodo de inicio tiene una h(S) muy alta y, a medida que va explorando ese valor ha de decrecer hasta una h(G) = 0.

También es importante decir que, como inicialización, se ha añadido el nodo de inicio en la lista Open. De esta manera será el primer nodo a explorar.

A partir de aquí el código iterará para cada nodo explorado. Por tanto, comprobaremos si la lista Open está vacía.

```
%3-check if openList is empty
if isempty(openList) == 1
    return
end
```

En esta iteración seleccionaremos el primer nodo de la lista de Open y lo pondremos en la lista de Closed para que no pueda ser explorado de nuevo, evitando así que se creen bucles de programa.

```
%4-select first node in openList and put it in closed. Remove
  from open
if iterations == 1
    closedList(1) = openList(1);
else
    closedList(end+1) = openList(1);
end
```

```
currentNode = openList(1);
openList(1) = [];
```

Acto seguido se realiza la comprobación de si el nodo actual es el nodo destino. Si es así, se devolverían las variables más importantes y acabaríamos proceso. Estas variables son el nodo actual que se está explorando, así como todos los nodos anteriores que conforman el camino desde el nodo de inicio, el path seguido y todos los nodos de las listas Open y Closed.

```
%5-if goal node, exit successfully
if currentNode.h == goalN.h
    disp("n == goalN");
    list_of_nodes = [closedList openList];
    solution_node = currentNode;
    previous_nodes = return_prev_nodes(currentNode,closedList);
    solution_path = currentNode.path;

    return
end
```

Es importante remarcar que la condición de éxito se realiza mediante h(n). Puesto que no se sabe cuántas iteraciones harán falta para encontrar una solución, nos parecía adecuado hacer las comprobaciones de éxito mediante este valor, puesto que sabemos cuánto ha de valer en el nodo final.

En caso de no ser el nodo destino, se expandirá este nodo. En este problema en concreto hemos definido la expansión de una manera peculiar. Puesto que cada nodo representa un lote de componentes, un nodo cualquiera se expande entre los componentes que se puedan pedir en ese problema. De esta manera, si solo existen 6 componentes posibles, el nodo actual se expandirá a 6 nodos. Si existen 9 componentes, a 9 nodos.

La función de expansión crea dichos nodos. Por lo tanto, el cálculo de la heurística ha de implementarse dentro de la expansión.

```
%6-expand
[expandedNodes, numNodes, id] = Expand(currentNode, components,
id, lista_com);
```

```
function [expandedNodes, numNodes, idOut] = Expand(currentNode,
components, id, lista_com)

numNodes = 0;

for i = 1:length(components)
```

```
    id = id + 1;
    newNode = Node;
    newNode.id = id;
    newNode.component = components(i);
    [newNode.g, newNode.h] = F_algorithm(newNode, currentNode,
lista_com);

    if newNode.h < 0
        newNode.h = 0;
    end

    switch i
        case 1
            newNode.path = currentNode.path+"A";
        case 2
            newNode.path = currentNode.path+"B";
        case 3
            newNode.path = currentNode.path+"C";
        case 4
            newNode.path = currentNode.path+"D";
        case 5
            newNode.path = currentNode.path+"E";
        case 6
            newNode.path = currentNode.path+"F";
        case 7
            newNode.path = currentNode.path+"G";
        case 8
            newNode.path = currentNode.path+"H";
        case 9
            newNode.path = currentNode.path+"I";
        otherwise
            newNode.path = currentNode.path+"$";
    end
    numNodes = numNodes + 1;
    expandedNodes(i) = newNode;
end

idOut = id;
end
```

El propósito de la función es el de asignar los valores intrínsecos de la clase Node (id, component, g y h, y path) para cada uno de los componentes disponibles en el problema.

La asignación de cada uno de ellos es trivial pero se le ha de dar especial atención a los valores de g(n) y de h(n). Para asignar esos valores se llama específicamente a una función llamada "F_algorithm". Esta implementará la heurística necesaria para estimar correctamente los valores necesarios de la función de evaluación. Puesto que es un paso tan importante, será explicado en detalle en el apartado siguiente.

Una vez se han expandido todos los nodos, para cada uno de ellos se realizan las siguientes acciones:

```
    %7-movie magic
    for c = 1:numNodes

        % busca si está en Closed
        isInClosed = isInList(expandedNodes(c),closedList);

        %si está en closed, pasa a la siguiente iteración
        if isInClosed == 1
            continue
        else

            % busca si está en open
            isInOpen = isInList(expandedNodes(c),openList);

            % si NO está en open, añadela
            if isInOpen == 0
                if isempty(openList) == 1
                    openList(1) = expandedNodes(c);
                else
                    openList(end+1) = expandedNodes(c);
                end
            end

        end
    end
```

Primero se busca si el nodo expandido está o no en la lista Closed. En caso de estar significaría que el nodo ya ha sido explorado, con lo que se desecha. Acto seguido comprueba que el nodo no esté en la lista Open. Si lo está no hace falta añadirlo de nuevo, por lo que el nodo se desecha.

Solamente en el caso de no encontrarlo en la lista Open se añade el nodo para su posterior evaluación en el algoritmo A*.

```
    %8-reordenar openList con valores ascendentes de f
    openList = ordenar(openList);

    %9-repeat
end
```

Finalmente se ordena la lista Open, utilizando el valor de f(n) de mayor a menor. De esta manera se asegura que, al iterar de nuevo el código, se escoja el nodo que más cerca está del nodo destino.

4 Algoritmo búsqueda heurística

Se ha realizado un algoritmo heurístico para complementar el A*. El objetivo de este algoritmo es el de encontrar el valor de h por cada nodo que explore. Este valor dependerá de la fecha de entrega de la comanda y de la capacidad del almacén.

La función que define el peso de cada nodo, como se explicó en el apartado anterior, es la siguiente:

$$f(n) = g(n) + h(n)$$

Dónde “g(n)” representa la distancia recorrida desde el nodo de inicio hasta el actual y “h(n)” representa la distancia aun por recorrer hasta el nodo final.

La función del algoritmo de búsqueda heurística recibe del A* la siguiente información:

- **Nodo actual**: nodo que está explorando el algoritmo A* y del cual nos pide su peso “h”.
- **Nodo previo**: nodo del que proviene el nodo actual, con todos sus predecesores.
- **Lista de comandas**: representa la lista de las comandas realizada por el usuario con las fechas de entrega para cada producto.

Con estos tres parámetros la función puede calcular un valor para el peso “h(n)”. La heurística del problema nos obliga a fijarnos en dos apartados que definirán cómo se calcula este peso:

- **Fecha de entrega**: cada una de las comandas de la lista tiene una prioridad de entrega, que es a su vez la máxima prioridad para el algoritmo A*. Por tanto, cómo se interpreten estos valores dentro del peso "h(n)" será de vital importancia.
- **Almacén**: A pesar que la heurística se basará principalmente en las fechas de entrega y las prioridades, también ha de tener en cuenta el estado del almacén. Un almacén sin capacidad de almacenar componentes no es deseable, y por tanto, tendrá que estar contemplado.

Sin embargo, antes de empezar a explicar la heurística definimos el valor de g(n).

Puesto que cada nodo ha sido interpretado como un pedido de un componente específico, una lista de nodos no es más que varios pedidos. Por lo tanto, el salto en

distancia de un nodo a otro será simplemente de 1. Eso genera que el valor de g(n) represente el nivel de profundidad del algoritmo.

A continuación se explica la heurística implementada y su funcionamiento en el código.

4.1 Peso "h" por fecha de entrega

El primer paso es ordenar la lista de las comandas ("lista_com") según las fechas de las comandas. En el siguiente ejemplo, donde se simula una lista de productos con sus prioridades en la columna de la derecha, se puede ver como se ordenan en función de la prioridad:

$$\begin{bmatrix} A & B & D & 9 \\ A & C & C & 15 \\ B & C & D & 5 \end{bmatrix} \rightarrow \begin{bmatrix} B & C & D & 5 \\ A & B & D & 9 \\ A & C & C & 15 \end{bmatrix}$$

Esta lista ordenada se utilizará durante el cálculo para tener en cuenta qué elementos se han pedido y cuales faltan por pedir. Durante la ejecución de este algoritmo se irán descontando los componentes pedidos hasta obtener una lista vacía.

A pesar que no tenga un peso importante en esta parte del algoritmo, se ha de mencionar que al ordenar la lista también se crea otra utilizando la función "modify_list". Esta segunda lista, inversa de la primera, será utilizada para recrear el almacén, que se explicará en apartados posteriores.

La metodología será la siguiente: el algoritmo lee la secuencia de componentes que se han pedido hasta el nodo actual. Por cada componente que ya se ha pedido se eliminan 4 componentes de la lista de comandas (puesto que los componentes se piden por lotes). El resultado es una lista de comandas con los componentes que aún faltan por pedir.

A continuación se muestra un ejemplo donde:

- Nodo actual: "C"
- Secuencia componentes pedidos: "A"

Lista de las comandas original:

$$\begin{bmatrix} A & B & C \\ C & C & A \\ A & B & D \end{bmatrix}$$

Lista modificada:

$$\begin{bmatrix} 0 & B & 0 \\ 0 & 0 & 0 \\ 0 & B & D \end{bmatrix}$$

Como se puede observar, se extraen los componentes pedidos previamente y también los del nodo actual.

Acto seguido se procede a evaluar la prioridad de cada componente. Puesto que la lista ha sido ordenada por prioridad, aquellos componentes que estén situados en las posiciones elevadas de la lista tendrán más prioridad. Por tanto, cada vez que aparezcan en la lista, se les sumará el siguiente valor:

$$k = 2^L$$

Dónde L es la longitud de la lista de comandas. El valor de k se irá reduciendo a la mitad cada vez que descendamos una fila, dándole así un valor más alto a las que tienen más prioridad.

Siguiendo el mismo ejemplo que antes, k adquiere los siguientes valores:

$$k = 2^3 = 8$$

$$\begin{bmatrix} 0 & B & 0 \\ 0 & 0 & 0 \\ 0 & B & D \end{bmatrix} \begin{matrix} k = 8 \\ k/2 = 4 \\ k/4 = 2 \end{matrix}$$

Por lo tanto, los componentes tendrán estos pesos:

$B = 8 + 2 = 10$

$D = 2$

Tal y como se ha implementado el algoritmo A*, el valor de h ha de decrecer a medida que se acerca al nodo final. Por consiguiente se han de invertir los valores obtenidos. Para hacerlo, se ha optado por sumar todos los valores de los componentes y restar el valor total a cada uno de ellos.

$$TotalG = GA + GB + GC + GD + GE + \cdots + GI$$

$$h_A = TotalG - GA$$

Siguiendo con el ejemplo obtenemos:

$$TotalG = GA + GB + GC + GD = 0 + 10 + 0 + 2 = 12$$

$$h_B = 14 - 10 = 4$$

$$h_D = 14 - 2 = 12$$

De esta manera sabemos que el componente B tiene preferencia respeto al componente D.

4.2 Peso "h" según capacidad del almacén

La función de almacén (descrita en el apartado 4) nos indica los espacios que se encuentran libres en el momento de analizar el nodo actual. A partir de ahí se calcula el valor "h_storage" (que dependerá de la capacidad del almacén) que se sumará al calculado previamente mediante las fechas de entrega.

Sin embargo, este peso solo se utiliza en el caso que el almacén esté saturado de componentes y no acepte ninguno más. Por lo tanto, "h_storage" tendrá un valor nulo excepto para el caso del almacén lleno, momento en que alcanzará un valor muy elevado (el valor de k para máxima prioridad). De esta manera se transmite al algoritmo A* que el camino que está recorriendo no cumple con las especificaciones y el insta a que busque un nuevo nodo.

5 Almacén

El objetivo principal del almacén es el de guardar los componentes pedidos hasta que el producto esté producido. Uno de los elementos a tener en cuenta a la hora de crear la heurística del algoritmo A* fue la de tener en cuenta el espacio disponible del almacén. Por tanto, como gestionar los elementos pedidos para cada uno de los nodos es la funcionalidad básica del código desarrollado.

5.1 Funcionalidad básica

El almacén ha de ser capaz de guardar los componentes que se vayan pidiendo, según al algoritmo vaya dando posibles soluciones. De esta manera se puede evaluar si la solución propuesta mantiene el almacén en un nivel estable o por el contrario lo llena muy rápidamente.

Dado que los pedidos de componentes han de hacerse en lotes de 4, cabe la posibilidad de tener componentes que no corresponden a ninguna comanda. Sin embargo, esos componentes han de poder guardarse en el almacén y deberán tenerse en cuenta cuando éste contenga los componentes que sí forman parte de los productos a fabricar.

El almacén ha de poder guardar los componentes, pero también ha de ser capaz de eliminar aquellos productos que ya han sido fabricados. Con este propósito se definen dos estados del producto:

- *producto reunido (assembled):* los componentes necesarios para el producto están en el almacén, pero aun no ha acabado su proceso de fabricación, modelado mediante una función normal.

- *producto finalizado (finished):* los componentes necesarios para el producto están en el almacén y ya ha acabado de fabricarse, por lo que se procede a extraerlo del almacén.

A pesar de que no sería estrictamente parte de la funcionalidad del almacén se ha decidido que una de las funcionalidades del almacén sería la de verificar los resultados temporales. Es decir, el almacén dispone de los tiempos de entrada de los componentes y de cuando el producto finaliza su fabricación, por lo que puede verificarse si esos tiempos entran dentro de las fechas de entrega impuestas en la lista de comandas.

5.2 Código

Para desarrollar la funcionalidad deseada se han creado tres ficheros. Dos de ellos, "**Storage**" y "**Product**" son clases destinadas a guardar los componentes, además de retener información de importancia que pueda ser necesaria.

Storage tiene 16 variables internas, llamadas posiciones. Cada una de esas posiciones refleja un espacio en el almacén que deberá llenarse con los componentes que crean un producto.

La clase Product, por tanto, tiene tres variables destinadas a cada uno de los posibles componentes que puedan almacenarse en su interior, llamadas "place". Además, la clase también dispone de cuatro variables dedicadas a retener información relevante: la id del producto ("asigned_request"), si los componentes guardados en una posición son de un producto de la lista de comandas o no ("is_product"), si los componentes de una posición han sido reunidos para empezar la fabricación ("is_assembled") y el tiempo necesario para fabricar el producto ("time_to_build").

```
classdef Storage

    properties
        pos1
        pos2
        pos3
        pos4
        pos5
        pos6
        pos7
        pos8
        pos9
        pos10
        pos11
        pos12
        pos13
        pos14
        pos15
        pos16
    end
```

```
classdef Product

    properties
        asigned_request
        is_product
        is_assembled
        time_to_build
        place1
        place2
        place3
    end
```

Por tanto, cada una de las 16 posiciones de la clase Storage estará ocupada por una clase Product, dando información sobre los componentes y el producto a fabricar almacenados.

Con estas clases definidas se crea una función llamada "store_management" que será la encargada de aplicar la funcionalidad deseada. Dicha función se llama dentro de la ejecución interna del programa cuando se está aplicando la heurística del algoritmo A*, y aparte de modelizar como quedaría el almacén con la información de ese nodo, el objetivo también es el de saber cuánto espacio queda en el almacén.

El código de la función se estructura en tres áreas:

- Inicialización de variables
- Asignación de componentes en el almacén
- Generación de listas

5.2.1 Inicialización de variables

Se inicializan las variables que se utilizaran en la función. En este apartado se crean 16 clases Product que se asignarán a cada una de las posiciones del Storage. De esta manera se puede empezar a asignar componentes dentro del almacén.

Además, utilizando el nodo examinado actualmente (puesto que la función se ejecuta mientras se evalúa un nodo) se consigue el orden de los pedidos en ese instante, necesario para poder almacenar los componentes correctamente. Es importante mencionar que, dado el momento en el que se ejecuta esta función el nodo evaluado aun no dispone de toda la información necesaria, también se ha de incluir el nodo previo (del cual se está expandiendo el actual).

Finalmente se inicializan las listas que se utilizarán más adelante.

5.2.2 Asignación de componentes en el almacén

Utilizando el orden de pedidos actualmente se empieza a almacenar los componentes dentro del almacén virtual. El algoritmo utilizado para tal fin se expone a continuación (en pseudocódigo):

```
for componente
      if está en lista de pedidos == true
            if producto ya en almacén == true
                  almacenar componente(pos. del producto)
            else
                  almacenar componente(primera pos. disponible)
            end
      else
            almacenar componente(última pos. disponible)
      end
end
```

Cada uno de los pedidos trae 4 componentes que deberán ser almacenados. Se realiza una búsqueda de dicho componente en la lista de pedidos. Existen dos posibilidades: que lo encuentre y que no.

En caso de encontrarlo, significa que ese componente es parte de un producto. Si este se halla ya dentro del almacén, el componente se sitúa en la posición asignada. En caso de no haber ningún otro componente de ese producto se le asigna la primera posición vacía que haya en el almacén.

Si el componente no está en la lista de pedidos es que dicho componente no forma parte de un producto y, por tanto, es un componente de más que sobra. Para

almacenar estos componentes no útiles se buscan en posiciones del almacén donde no haya productos.

Por tanto, los componentes que formen parte de algún producto estarán situados en las primeras posiciones del almacén, mientras que los descartes estarán en el otro extremo.

5.2.3 Generación de listas

Una vez asignados todos los componentes en el almacén, independientemente de si son parte de un producto o no, se generan las listas de resultados.

Existen tres listas, una para comprobar los productos del almacén que tienen la condición "assembled" (los componentes están reunidos y se empieza la fabricación), otra lista para comprobar si los productos tienen la condición "finished" (el producto ha sido fabricado y por tanto debe salir del almacén) y una tercera lista de resultados que compara tiempos y comprueba que el producto ha sido fabricado dentro del plazo acordado.

Las dos primeras listas (assembled_product_list y finished_product_list) son sencillas de interpretar.

La primera lista revisa los componentes de cada una de las posiciones del almacén y si encuentra tres componentes del mismo producto que no tengan la condición "is_assembled", se la asigna. La lista entonces guarda los datos más importantes, como los propios componentes (columnas 3, 4 y 5), la identificación del producto (columna 2), el tiempo en el que se ha detectado (columna 1) así como el tiempo que necesita para fabricarse (columna 6).

assembled product list

"t"	"req"	"p1"	"p2"	"p3"	"time_to_build"
"3"	"2"	"A"	"B"	"F"	"2.0376"
"4"	"1"	"G"	"A"	"B"	"0.6641"
"5"	"3"	"A"	"G"	"I"	"1.4685"
"6"	"4"	"B"	"C"	"F"	"0.921"
"8"	"7"	"C"	"D"	"I"	"1.6954"
"8"	"5"	"G"	"D"	"E"	"1.3279"
"9"	"6"	"E"	"F"	"H"	"1.5905"

La lista de productos acabados revisa la lista anterior. La suma del tiempo en que se empezó su fabricación y el que requiere para poder fabricarse será el momento en que ese producto esté finalizado. En cuanto el tiempo de simulación sea superior al de finalizado, ese producto estará completado y deberá salir del almacén. Los datos

más importantes se guardarán dentro de la lista "finished_product_list" para tener constancia de que el producto ha sido fabricado correctamente.

finished product list

"t"	"req"	"p1"	"p2"	"p3"
"6"	"2"	"A"	"B"	"F"
"5"	"1"	"G"	"A"	"B"
"7"	"3"	"A"	"G"	"I"
"7"	"4"	"B"	"C"	"F"
"10"	"7"	"C"	"D"	"I"
"10"	"5"	"G"	"D"	"E"
"11"	"6"	"E"	"F"	"H"

La última lista es la de resultados. Esta simplemente recoge los valores de tiempos de las dos listas anteriores. Para cada producto finalizado, evalúa si el tiempo en el que se ha completado el producto es inferior al impuesto al hacer la comanda.

results list

"prod"	"as_t"	"time_b"	"f_t"	"deadline"	"IN TIME"
"2"	"3"	"2.0376"	"6"	"5"	"false"
"1"	"4"	"0.6641"	"5"	"10"	"true"
"3"	"5"	"1.4685"	"7"	"6"	"false"
"4"	"6"	"0.921"	"7"	"6"	"false"
"7"	"8"	"1.6954"	"10"	"9"	"false"
"5"	"8"	"1.3279"	"10"	"8"	"false"
"6"	"9"	"1.5905"	"11"	"9"	"false"

Las cuatro primeras columnas simplemente vuelven a destacar información que ya se ha dado, como qué producto es (columna 1), los tiempos de reunión de componentes y el necesario para fabricar el producto (columnas 2 y 3) y el tiempo de finalización (columna 4). Este último valor se compara con el deadline, la fecha indicada en la lista de comandas inicial. Por tanto, si entra dentro de lo establecido, la última columna lo reflejará.

Es posible que un producto no se fabrique a tiempo. Esto puede venir determinado por diversos motivos: la distribución normal del tiempo de fabricación depende de la aleatoriedad, así que es completamente posible que se obtenga un valor muy elevado que sumado a unas deadlines muy ajustadas, haga que producto no se finalice a tiempo. Para ilustrar este caso se puede observar el ejemplo mostrado en la lista, en donde hay varios productos que no se fabrican a tiempo.

6 Resultados

Para poder analizar correctamente los resultados, se ha creado una lista de comandas con prioridades ajustadas.

En este ejemplo, las comandas son: ABC, BCB, AEB, DBA y AED, y las fechas de entrega son: 7, 9, 6, 8 y 5 (respectivamente). Para este problema no se utilizarán los 9 componentes, sino solamente los 6 primeros (de "A" a "F"). La lista ordenada por prioridad es la siguiente:

"A" "E" "D" "5"

"A" "E" "B" "6"

"A" "B" "C" "7"

"D" "B" "A" "8"

"B" "C" "B" "9"

Al ejecutar el programa se obtiene la siguiente secuencia de componentes a pedir:

SAEDBCB

Siendo S el nodo inicial. Analizando solamente mediante las prioridades se puede confirmar que el algoritmo A* obtiene una secuencia de componentes correcta. Sin embargo, aun falta comprobar si los productos han sido fabricados dentro del plazo establecido.

El algoritmo también devuelve información importante a analizar. En este caso, devuelve el nodo final y la secuencia de nodos desde el inicio hasta dicho nodo.

Nodo final:

id: 33
component: "B"
g: 7
h: 0
path: "SAEDBCB"

Lista de nodos previa:

id: 28
component: "C"
g: 6
h: 2
path: "SAEDBC"

id: 21
component: "B"
g: 5
h: 10
path: "SAEDB"

id: 17
component: "D"
g: 4
h: 42
path: "SAED"

id: 12
component: "E"
g: 3
h: 78
path: "SAE"

id: 2
component: "A"
g: 2
h: 126
path: "SA"

id: 1
component: "S"
g: 0
h: 10000
path: "S"

Como se puede comprobar, la lista de nodos previa va desde el nodo final hasta el inicial. El valor de h de cada nodo va descendiendo a medida que el algoritmo A* va acercándose a la solución, hasta que es finalmente 0 en el nodo final.

De esta manera se conoce el camino obtenido por el programa de búsqueda heurística. Se puede ver de una manera más representativa en el siguiente gráfico:

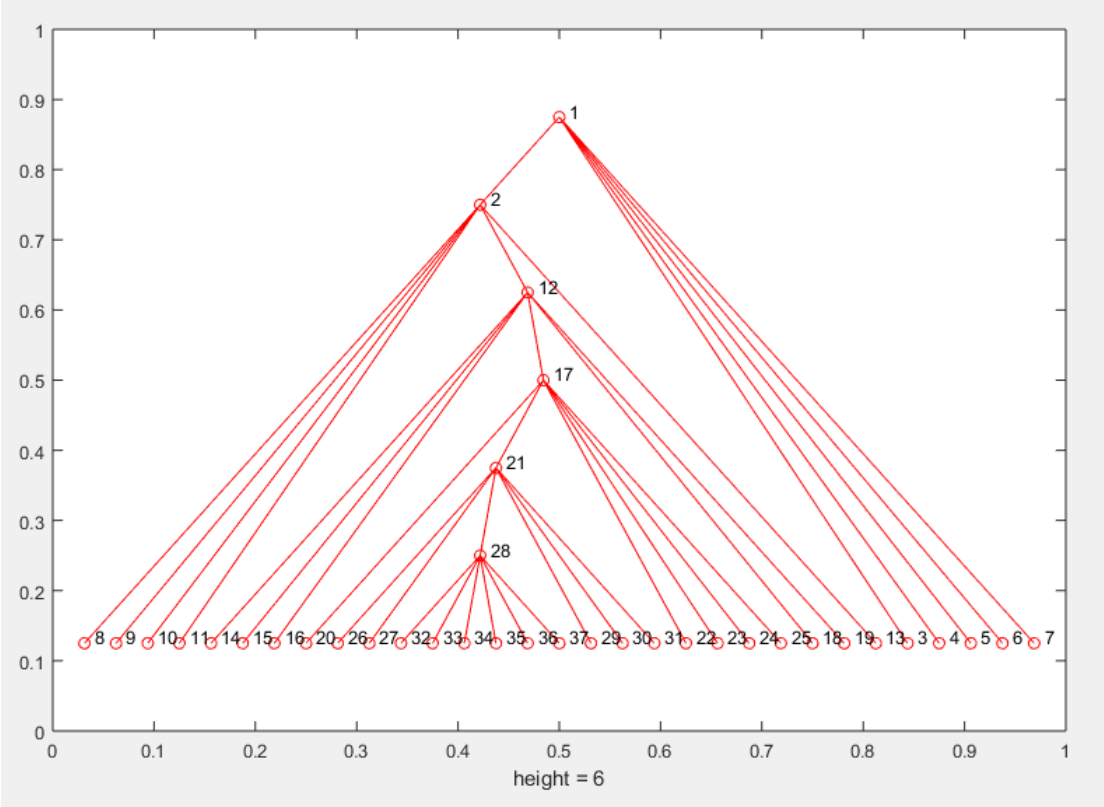

Ilustración 2 "Tree representation" de los nodos utilizados en el problema

En este gráfico se pueden ver todos los nodos utilizados por el algoritmo, e incluso el camino que el programa sigue para localizar el nodo con menor valor de f(n)=g(n)+h(n).

En este caso el programa encuentra los componentes necesarios en los nodos 1, 2, 12, 17, 21, 28 y 33. Como se puede observar, en este problema el programa ha comprobado hasta 37 nodos hasta poder llegar al final. El primer nodo se divide en los nodos 2, 3, 4, 5, 6 y 7, cada nodo en el que encuentra el menor valor de “f” se divide en 6 nodos ya que es el número de componentes que existen en esté problema (A, B, C, D, E, F).

Si pasamos a analizar el almacén, este muestra qué hay actualmente (en el momento de finalizar el algoritmo) en cada una de sus posiciones. Se observa como el almacén queda vacío de los productos finalizados, pero se quedan los

componentes sobrantes de los lotes que ha pedido, los cuales se almacenan en la parte final de la lista:

Storage

"p1"	"p2"	"p3"	"a_req"	"is_pr"	"is_as"	"time"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"_"	"_"	"_"	"_"	"_"	"_"	"_"
"B"	"B"	"B"	"0"	"false"	"_"	"_"
"C"	"C"	"D"	"0"	"false"	"_"	"_"
"D"	"E"	"E"	"0"	"false"	"_"	"_"

Por lo tanto, se puede decir que en este caso nos sobrarían tres componentes de B, dos componentes de C, un componente D y dos componentes de E.

También es importante comentar las listas obtenidas.

assembled product list

"t"	"req"	"p1"	"p2"	"p3"	"time_to_build"
"3"	"1"	"A"	"E"	"D"	"0.82922"
"4"	"2"	"A"	"E"	"B"	"1.9569"
"4"	"4"	"D"	"B"	"A"	"0.068445"
"5"	"3"	"A"	"B"	"C"	"1.4635"
"6"	"5"	"B"	"C"	"B"	"1.4811"

En la lista de "assembled" podemos observar el momento en que se alcanzó dicha condición (columna 1) para qué producto (columna 2). También se pueden ver los componentes del producto y el tiempo necesario para fabricarlo (siguiendo una distribución normal).

Los tiempos en este caso son relativamente bajos puesto que muchos productos comparten componentes y, al pedir los componentes en lotes, se reúnen más rápidamente.

```
finished product list
    "t"    "req"    "p1"    "p2"    "p3"
    "4"    "1"      "A"     "E"     "D"
    "5"    "4"      "D"     "B"     "A"
    "6"    "2"      "A"     "E"     "B"
    "7"    "3"      "A"     "B"     "C"
    "8"    "5"      "B"     "C"     "B"
```

En la lista de "finished" se pueden ver el momento en que se acabó de fabricar (columna 1) y qué producto (columna 2). Puesto que los productos acabados salen del almacén en cuanto se fabrican, aquí se puede constatar que en efecto los productos se completaron correctamente.

Finalmente, en la lista de resultados se puede ver si los productos han sido fabricados dentro del tiempo establecido o no.

```
results list
    "prod"    "as_t"    "time_b"      "f_t"    "deadline"    "IN TIME"
    "1"       "3"       "0.81436"     "4"      "5"           "true"
    "2"       "4"       "1.8561"      "6"      "6"           "true"
    "4"       "4"       "1.0016"      "6"      "8"           "true"
    "3"       "5"       "1.0358"      "7"      "7"           "true"
    "5"       "6"       "2.3022"      "9"      "9"           "true"
```

Como puede comprobarse, los 5 productos de este ejemplo han sido completados a tiempo. Sin embargo, ha de mencionarse que no siempre es así. Dado que el tiempo de fabricación es un valor de distribución normal, puede darse el caso de tener unos tiempos de fabricación muy grandes y, si la fecha de entrega es muy ajustada, no cumplir con las especificaciones.

No obstante, con esto se demuestra que el algoritmo funciona y es capaz de dar una solución viable y eficiente.

7 Conclusiones

El algoritmo de búsqueda heurística cumple los requisitos impuestos y es capaz de devolver una secuencia de los componentes en el orden óptimo para conseguir crear los productos con unas fechas de entrega determinadas y sin llenar el almacén.

A su vez, se ha ahondado en el conocimiento de los algoritmos de búsqueda, especialmente con el algoritmo A*, utilizando una heurística diseñada para encontrar la respuesta a una comanda con unas ciertas condiciones. Por lo tanto, es posible

solucionar problemas con distintas condiciones y parámetros utilizando el algoritmo A*.

Se ha de mencionar, aun el éxito del resultado, que es posible que se pueda encontrar una mejor heurística que defina el problema. Puesto que las condiciones iniciales especificaban una lista estática de productos (que no se iba actualizando) y un número finito de componentes, nuestra heurística funcionaba bien, pero si se cambian algunas condiciones no podemos asegurar su éxito. En ese caso habría de replantearse el problema y adaptar la solución a las nuevas necesidades.

8 Bibliografia

1. Nils J Nilsson. *The quest for artificial intelligence a history of ideas and achievements.* Cambridge University Press, 13 Setembre 2009. DOI: http://ai.stanford.edu/~nilsson/QAI/qai.pdf

   L'apartat *12.1.1 A∗ : A New Heuristic Search Method (pag. 216)* explica el métode de cerca heurística A*.

2. Kramer Oliver. *Self-Adaptive Heuristics for Evolutionary Computation.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. DOI: https://link-springer-com.recursos.biblioteca.upc.edu/content/pdf/10.1007%2F978-3-540-69281-2.pdf

   2. Evolutionary Algorithms
   3. Self-Adaptation

3. Elaine Rich, Kevin Knight. *Inteligencia Artificial.* McGraw-Hill Publishing Co., 1990.

   Explicació dels métodes de cerca heurística, amb especial atenció a l'apartat de l'algoritme A*.

4. Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Springer Science, 2008. DOI: http://www.math.nsc.ru/LBRT/k5/Scheduling/Scheduling_Theory,%20Algorithms,%20and%20Systems(Pinedo,2008).pdf

5. Constantino Malagón. *Búsqueda heurística*. DOI: https://www.nebrija.es/~cmalagon/ia/transparencias/busqueda_heuristica.pdf
6. Pedro Meseguer. *Búsqueda Heurística I*. Bellaterra. DOI: http://www.iiia.csic.es/~pedro/busqueda1-introduccion.pdf

7. Fernando Berzal. *Búsqueda en Inteligencia Artificial*. Universidad de Granada. DOI: http://elvex.ugr.es/decsai/intelligent/slides/ai/A2%20Search.pdf
8. *Búsqueda Heurística*. UPC 2011/2012. DOI: http://www.lsi.upc.es/~bejar/ia/transpas/teoria/2-BH2-Busqueda_heuristica.pdf
9. Peter E. Hart, Nils J. Nilsson, Bertram Raphael. *A Formal Basis for Heuristic Determination of Minimum Cost Paths*.
10. Nilsson, N. J. *Artifitial Intelligence. A new synthesis.* DOI: http://sci.neyshabur.ac.ir/sci/images/electricalengineering/ebook/Nils%20J%20Nilsson%20Artificial%20Intelligence%20A%20New%20Synthesis%20-%201998.pdf