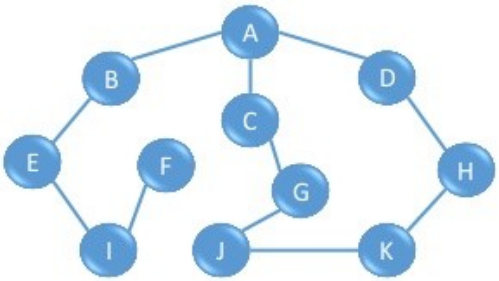
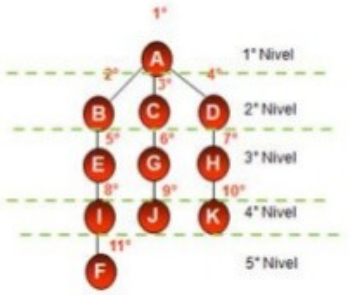
º A continuación voy a narrar cómo he resuelto los distintos niveles de la práctica 2 (los extraños mundos de Belkan).

- Nivel 0: este es el nivel demo, y por ello no tiene gran dificultad. El algoritmo pathfinding viene ya implementado y lo único que debemos hacer es conseguir que Belkan se mueva por medio de la función think. Al ser el nivel base y estar detallado en el tutorial no voy a centrarme en describir la implementación.

- Nivel 1: para este nivel se nos pide implementar un algoritmo que haga un camino para Belkan óptimo en número de acciones. Claramente vamos a necesitar un pathfinding en anchura, y la única diferencia notable con respecto al algoritmo en profundidad creado en el nivel 0, es que vamos a utilizar una cola en vez de una pila, debido a que debemos recorrer el árbol de decisiones por niveles, teniendo en cuenta cada rama.




• Recorrido en anchura desde vértice A = {A-B-C-D-E-G-H-I-J-K-F}


- Nivel 2: se nos pide encontrar un camino al objetivo que sea óptimo en batería. Para este nivel he implementado un algoritmo con costo uniforme. Aquí hay que tener varias cosas en cuenta, debido a que cada casilla tiene un costo de batería distinto en función no solo de la propia casilla, sino también del objeto que lleves y de la acción que realices. Debido a esto he creado un método llamado costeCasilla donde tengo en cuenta la acción que se realiza (forward, turnleft o turnright), la casilla en la que está (agua, bosque…) y si tiene zapatillas o bikini (lo cual reducirá el coste dependiendo de la casilla en la que se encuentre. Los costes de la casilla se almacenan en la variable costCasilla dentro del estado del nodo y después se acumulan al coste uniforme (también guardado

en estado) que tengan estos nodos (este coste es la representación del coste total del camino que han recorrido los nodos ancestros de ese nodo en concreto).
Ahora que tenemos como calcular los costes, necesitamos saber como almacenarlos. Para ello he utilizado una cola con prioridad que ordene los costes (así sacaremos los nodos que representen caminos menos costosos en batería). Para ordenar la cola he utilizado una comparación entre costes uniformes (costes acumulados) de los nodos que se guarden.
Otra cosa que tenemos que tener en cuenta es que los nodos deben coger bikini o zapatillas si las encuentran. Para ello utilizo una función que mira el mapaResultado y si hay objeto lo coge (eso sí, suelta el otro objeto si es que tenía). Las variables utilizadas son dos booleanos, uno para zapatillas y otro para bikini.
Además de lo anterior, también debemos considerar como un estado distinto un nodo que sea igual a otro en todos sus atributos, pero que difiera en cuanto a objetos. Simplemente modificamos comparaEstados añadiendo más opciones que hagan diferente a ese estado del nodo, así serán diferentes dos estados igual en fila, columna, orientación, pero que posean distintos objetos o que no tengan un objeto en un momento dado y luego sí. Esto nos permite trazar planes con más opciones de movilidad para Belkan.
Otra implementación que he realizado y que tiene que ver con la eficiencia del programa es mantener los nodos repetidos en abiertos (esto se debe a que se puede dar el caso de tener nodos mucho menos costosos que otros, pero que por el simple hecho de ser iguales no se tienen en cuenta). Cuando hacemos pop de la cola compruebo que el nodo no esté en cerrados.
En cuanto a la expansión de nodos hijos, expando los nodos de acciones turn_left, turn_right y forward seguidos. En cada uno de ellos calculo el coste de casilla y lo acumulo junto con el coste uniforme que hayan ido heredando de los nodos padres. Esto se mantiene para todos los niveles superiores.

- Nivel 3: para este nivel se nos pide lo mismo que en el 2 con el aliciente de que son 3 objetivos los que debemos alcanzar en vez de solo 1. He repetido algoritmo (costoUniforme) y la única modificación significativa es que se termina de buscar cuando se hayan encontrado 3 objetivos, los cuales represento como un vector de booleanos dentro del estado de cada nodo. Igual que hicimos con bikini y zapatillas, aquí he considerado que dos nodos son distintos si uno de ellos tiene el vector de objetivos distinto del otro (pues pueden ser iguales en el resto de atributos, pero que uno de ellos haya encontrado en su camino 2 objetivos y el otro ninguno).

- Nivel 4: para este nivel se nos pide encontrar el máximo número de objetivos posible en un mapa sin explorar, es decir, Belkan no tiene conocimiento alguno sobre las casillas que le rodean, por tanto deberá explorar y adaptar el plan elegido. Para este nivel repito algoritmo con costo uniforme (nivel 2).
Lo primero es explorar casillas y actualizar el mapa, para ello, he utilizado una función auxiliar llamada rellenarMapa. Dependiendo del sentido en el que se encuentre Belkan deberá rellenar distintas casillas del mapa, por ello distingo entre los 4 sentidos posibles y actualizo el mapa con el sensor de terreno. Esta función se llamará cada vez que lo haga think, es decir, a cada acción que realice Belkan se actualizará el mapa delante suya.
También he utilizado una función calcularNuevoObjetivo, la cuál calcula la distancia entre Belkan y los objetivos que haya en ese momento para elegir al objetivo que esté más cerca de este. Esta función se llamará cada vez que se llame a pathfinding para encontrar un nuevo plan (por lo que siempre se traza un plan al objetivo más cercano).
Se debe calcular una nueva ruta siempre que Belkan dé con un muro, con un precipicio o cuando quiera meterse dentro de zonas de agua o bosque sin su objeto correspondiente.

Para los aldeanos simplemente he hecho que Belkan espero quieto (lo cuál no consume batería) hasta que este se aparte de su camino. Esta forma de evitarlos es eficiente en cuanto a batería y funciona bastante bien, los aldeanos se suelen apartar bastante rápido.
Para poder maximizar los objetivos que Belkan alcance, he hecho que si se encuentra una batería se mantenga quieto hasta recargarse (siempre y cuando sus niveles sean inferiores a cierto umbral de batería).
La última cosa y la más importante que he tenido en cuenta es que cada vez que alcancemos un objetivo, debemos borrarlo de la lista de objetivos activa, pues ya lo hemos cogido. Belkan entonces trazará un nuevo plan al siguiente objetivo más cercano (escogido con la función calcularNuevoObjetivo). La lista de objetivos no será actualizada con nuevos hasta que se vacíe la anterior, dicho de otra forma, hasta que se alcancen todos.