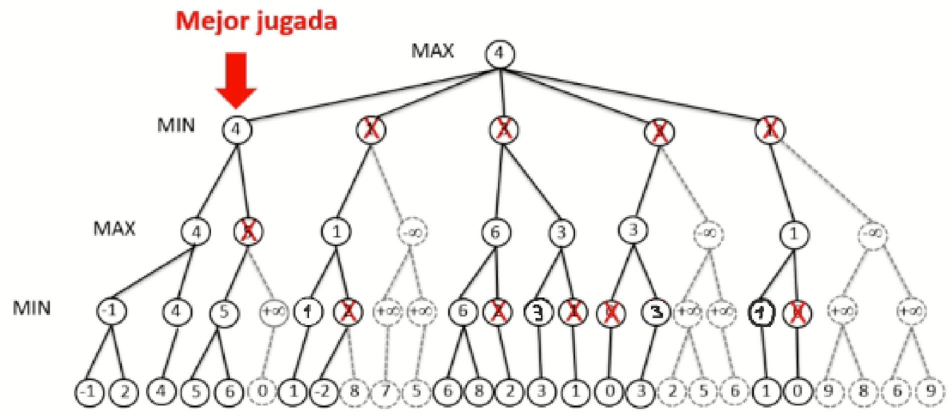
```
(deffacts datos
  (puerto A E B B C D B B A E D plataforma T1 grua T1 vacio plataforma T2 grua T2 vacio)
  (contenedor T1 A D)
  (contenedor T2 B C E)
)

;; al principio cree una pila vacia de un contenedor existente si no hay nada en su plataforma ni en su grua
(defrule pila_inicial
  (puerto $?p1 ?p $?p2 plataforma ?t $?l grua ?t vacio $?r)
  (test (not (member$ ?p $?l)))
  (contenedor ?t $? ?p $?)
  =>
  (assert (puerto $?p1 ?p $?p2 plataforma ?t $?l pila ?p 0 grua ?t vacio $?r))
)

;;coge un contenedor solo si su grua está vacia y hay una pila no completa (menos de3)
(defrule coge_contenedor
  (puerto $?p1 ?p $?p2 plataforma ?t $?e1 pila ?p ?c $?e2 grua ?t vacio $?r)
  (contenedor ?t $? ?p $?)
  (test (< ?c 3))
  =>
  (assert (puerto $?p1 $?p2 plataforma ?t $?e1 pila ?p ?c $?e2 grua ?t ?p $?r))
)

;;la grua deja un contenedor y no llena la pila o no quedan más de esa clase
(defrule parada
  (declare (salience 100))
  (puerto plataforma T1 $?p1 grua ? vacio plataforma T2 $?p2 grua ? vacio )
  =>
  (printout t "el número de pilas es " (+ (div (length$ $?p1) 3)(div (length$ $?p2) 3)) crlf)
  (halt)
)
------------------------------------------------------------
(defrule empujar-dcha
  (juego pos ?x ?y $?pre 1 ?lx ?y $?post nivel ?n)
  (limites-x ?min ?max)
  (test (< (+ ?x 1) ?max))
  (profundidad-maxima ?prof)
  (test (< ?n ?prof))
  (not (contenedor =(+ ?x 2) ?y))
  (test (not (member$ (create$ 1 (+ ?x 2) ?y) $?pre)))
  (test (not (member$ (create$ 1 (+ ?x 2) ?y) $?post)))
  (test (- (+ ?x 1) ?lx))
  =>
  (bind ?*nod-gen* (+ ?*nod-gen* 1))
  (assert (juego pos ?lx ?y $?pre 1 (+ ?lx 1) ?y $?post nivel (+ ?n 1)))
)
```

Criterio	Anchura	Coste uniforme	Prof.	Prof. Iterativa
Temporal	$O(b^{d+1})$	$O(b^{C^*/\varepsilon})$	$O(b^m)$	$O(b^d)$
Espacial	$O(b^{d+1})$	$O(b^{C^*/\varepsilon})$	$O(b.m)$	$O(b.d)$
Optima?	Sí *	Sí	No	Sí *
Completa?	Sí	Sí **	No	Sí

* Óptima si los costes de las acciones son todos iguales

** Completa si los costes de las acciones son $\geq \varepsilon$ para un ε positivo



BÚSQUEDA VORAZ.

La **búsqueda voraz** expande el nodo que parece estar más cerca del objetivo ya que, probablemente, dicho nodo conduce más rápidamente a una solución.

Evalúa nodos utilizando simplemente $f(n) = h(n)$.

- **Expande el nodo más cercano al objetivo.**
- Búsqueda *primero-el-mejor* voraz.

EVALUACIÓN.

- **COMPLETA.**
  - → NO, se puede quedar estancado en un ciclo.
  - → Se asemeja a primero en profundidad (prefiere seguir un camino único al objetivo).
  - → La versión GRAPH-SEARCH es completa.
- **ÓPTIMA.**
  - → No, en cada paso escoge el nodo más cercano al objetivo (voraz).
- **COMPLEJIDAD TEMPORAL.**
  - → $O(b^m)$ donde m es la profundidad máxima del espacio de búsqueda.
  - → La utilización de buenas heurísticas puede mejorar la búsqueda notablemente. La reducción del espacio de búsqueda dependerá del problema particular y la calidad de la heurística.
- **COMPLEJIDAD ESPACIAL.**
  - → $O(b^m)$ donde m es la profundidad máxima del espacio de búsqueda.

- **H2: distancias de Manhattan**
  - ✓ h2(n): distancias de Manhattan (suma de las distancias de cada ficha a su posición objetivo)
  - ✓ Elimina Restricción 2
  - ✓ Una ficha se puede mover a cualquier casilla adyacente
  - ✓ h2(n) devuelve una estimación optimista (solución óptima para el problema relajado
  - ✓ h2(n)=1+2+4+1+1=9 para el ejemplo del estado inicial

Distancias de Manhattan domina a fichas descolocadas (h2(n)>= h1(n), ∀n)

ALGORITMO MINIMAX (*John von Neumann*).

1. Generar el árbol de juego hasta un cierto nivel de profundidad en anchura hasta alcanzar los nodos terminales en el nivel máximo de profundidad.
2. Aplicar la función de utilidad a cada nodo terminal.
3. Usar los valores de utilidad de los nodos terminales para determinar la utilidad de los nodos del nivel superior. *Volcar* los valores de utilidad desde los nodos hoja hasta la raíz, nivel por nivel (profundidad). En los nodos MIN se vuelca el menor valor de utilidad de sus sucesores; en los nodos MAX se vuelca el mayor valor de utilidad de sus sucesores.

— $\boldsymbol{Valor\_Minimax(n)}$ =

→ $\boldsymbol{Utilidad(n)}$ si n es un nodo terminal.

→ $\boldsymbol{max_{s\epsilon Sucesor(n)}\ Valor\_Minimax(s)}$ si n es un nodo MAX.

› De las distintas jugadas que puede realizar un nodo MAX, la mejor será la que *le llevé* al nodo hijo con mayor valor de la función de utilidad.

→ $\boldsymbol{min_{s\epsilon Sucesor(n)}\ Valor\_Minimax(s)}$ si n es un nodo MIN.

› De las distintas jugadas que puede realizar un nodo MIN, la mejor será la que *le llevé* al nodo hijo con menor valor de la función de utilidad.

4. Eventualmente, el nodo raíz adquiere un valor volcado; en este punto, MAX escoge el movimiento que le conduce al estado de máxima utilidad.
   En el caso de producirse empates entre los máximos valores de utilidad de algunos hijos del nodo raíz, da igual cuál de ellos elijamos como mejor jugada. Cuando se produce uno de estos empates, decimos que se produce una *meseta*.

EFICIENCIA MINIMAX/ $\alpha - \beta$.

En un $\alpha - \beta$ óptimo se generarán $O(b^{d/2})$ mientras que en Minimax se generan $O(b^d)$,

PODA.

— **Corte α: β provisional $\leq \alpha$ predecesor** (no sólo padre).
Si un nodo A tiene un valor provisional α mayor o igual que el valor provisional β de su hijo B, B no necesita seguir profundizando en sus nodos.

— **Corte β: α provisional $\geq \beta$ predecesor** (no sólo padre).
Si un nodo A tiene un valor provisional β menor o igual que el valor provisional α de su sucesor B, B no necesita seguir profundizando en sus nodos.

PROCESO GENERAL DE BÚSQUEDA.

1. nodo-acutal ← estado inicial del problema.
2. Comprobar si nodo-actual es el estado final del problema.
3. Aplicar las acciones del problema en dicho estado y generar el conjunto de estados.
4. Escoger un no que no ha sido expandido todavía (nodo-actual).
5. Volver al paso 2.

El conjunto de nodos no expandidos se denomina **conjunto frontera**, **nodos hoja** o **lista OPEN**.

BÚSQUEDA EN ÁRBOL.

1. Introducir el estado inicial en la lista OPEN.
2. Escoger (y eliminar) nodo de la lista OPEN (expansión del nodo).
3. Comprobar si dicho nodo es el estado final del problema.
4. Generar hijos y añadirlos a la lista OPEN.
5. Volver al paso 2.

BACKTRACKING.

Se almacena una lista PATH para aplicar *backtracking*: almacena los nodos expandidos del camino actual y se eliminan cuando se llama a la función *backtracking*.

Algoritmo:

1. Eliminar $\boldsymbol{n}$ de la lista PATH.
   → Poner $\boldsymbol{n}$ en la lista PATH.
   → Comprobar si $\boldsymbol{n}$ es objetivo: NO.
   → Comprobar si $\boldsymbol{n}$ está en el máximo nivel de profundidad: SÍ.
2. Si parent(n) no tiene más hijos en OPEN $\Longrightarrow$ BACKTRACKING(parent(n)).
3. Si parent(n) tiene más hijos en OPEN $\Longrightarrow$ escoger siguiente nodo de la lista OPEN.

PROBAR QUE SI $h(n)$ ES ADMISIBLE, ENTONCES A ES ÓPTIMO.*

$\boldsymbol{G}$ es un estado objetivo óptimo: $\boldsymbol{f(G) = g(G)}$.

$\boldsymbol{n}$ es un nodo en la lista OPEN en el camino óptimo a $\boldsymbol{G}$.

$$f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G) = f(G) \Longrightarrow \boldsymbol{f(n) \leq g(G)}$$

$\boldsymbol{G_{no}}$ es un estado subóptimo en la lista OPEN con coste

$$\boldsymbol{f(G_{no}) = g(G_{no}) > g(G)}.$$

Si no se coge $\boldsymbol{n}$ para expansión es porque $f(n) \geq f(G_{no})$.

Por tanto

$$f(G_{no}) \leq f(n) \leq g(G) \Longrightarrow g(G_{no}) \leq g(G)$$

Alcanzar esta contradicción implica que se escoge $\boldsymbol{n}$.

ALGORITMO $\alpha - \beta$.

VALOR DEFINITIVO/VALOR PROVISIONAL.

— El valor de un nodo es **definitivo** cuando
  → es un nodo terminal o
  → ya han sido explorados todos sus descendientes (y volcado sus valores en él cuando sea oportuno).

— El valor de un nodo es **provisional** cuando
  → no es un nodo terminal y
  → no han sido explorados todos sus descendientes (y volcado sus valores en él cuando sea oportuno).

1. Generar el árbol de juego hasta un cierto nivel de profundidad en anchura.
2. Asumir que los nodos MAX tienen un valor volcado inicial $\alpha = -\infty$ y los nodos MIN tienen un valor volcado inicial $\beta = +\infty$.
3. Generar un nodo terminal en profundidad y calcular su valor de utilidad.
4. Volcar dicho valor sobre el nodo padre provisionalmente, si la utilidad *relativa* es mayor (mantener el mejor valor).
5. Comprobar condición de corte; al volcar un valor a un nodo, este valor se compara con los valores de todos los nodos predecesores contrarios con objeto de ver si se produce un corte α o β.
   — Corte α: β provisional $\leq \alpha$ predecesor (no sólo padre).
   — Corte β: α provisional $\geq \beta$ predecesor (no sólo padre).
   Si se produce un corte, regresar a 4.
6. Si el valor volcado es un valor provisional, regresar a 3. Si el valor volcado es un valor definitivo, regresar a 4.
   — Si es un nodo MAX, su nuevo valor será el máximo del valor que tiene y el de su hijo.
   — Si es un nodo MIN, su nuevo valor será el mínimo del valor que tiene y el de su hijo.
   *Si un nodo tiene más de un hijo y estos tienen valor definitivo, se volcará sobre el padre el valor de cada uno de los hijos de forma iterativa siempre y cuando el valor provisional del padre tenga menor utilidad *relativa* (menor en caso de MAX, mayor en caso de MIN) que el valor definitivo del hijo.
7. Eventualmente, el nodo raíz adquiere un valor volcado; en este punto, MAX escoge el movimiento que le conduce al estado de máxima utilidad.
   En el caso de producirse empates entre los máximos valores de utilidad de algunos hijos del nodo raíz, da igual cuál de ellos elijamos como mejor jugada. Cuando se produce uno de estos empates, decimos que se produce una *meseta*.
   *Utilidad *relativa*: nuevo valor de utilidad mayor que el anterior (para MAX) y nuevo valor de utilidad menor que al anterior (para MIN).

PROFUNDIDAD ITERATIVA.

$\boldsymbol{function}$ $Iterative\_Deepening\_Search\ (problem)$ $\boldsymbol{returns}$ $(solution, failure)$
$\boldsymbol{inputs}$: $problem$ /*$a\ problem$*/
$\boldsymbol{for}$ $depth = 0$ $\boldsymbol{to}$ ∞ $\boldsymbol{do}$
$result = dept\_limited\_search(problem, depth)$
$\boldsymbol{if}$ $result <> failure$ $\boldsymbol{return}$ $result$
$\boldsymbol{end}$
$\boldsymbol{return}$ $failure$
$\boldsymbol{end\ function}$

PROPIEDADES.

— **COMPLETA.**
— **ÓPTIMA.**
→ Cuando anchura lo es.
— **COMPLEJIDAD TEMORAL.**
→ $O(b^d)$.
— **COMPLEJIDAD ESPACIAL.**
→ $O(b \times d)$.

COMPARACIÓN A* CON OTRAS ESTRATEGIAS.

— Anchura: equivalente a $f(n) = nivel(n) + 0$, donde $h(n) = 0 < h^*(n)$.
— Coste uniforme: equivalente a $f(n) = g(n) + 0$, donde $h(n) = 0 < h^*(n)$.
— Profundidad: no comparable a A*.

CONOCIMIENTO HEURÍSTICO.

— $h(n) = 0 \rightarrow$ ausencia de conocimiento.
— $h(n) = h^*(n) \rightarrow$ conocimiento máximo.
— Si $h2(n) \geq h1(n) \forall n$ (ambos admisibles) entonces $h2$ domina a $h1$ ($h2$ es más informado que $h1$); $h2$ nunca expandirá más nodos que $h1$.

ALGORITMO GRAPH-SEARCH.

— La lista OPEN se implementa como una cola de prioridades (priority queue): se extrae el elemento de la cola con la máxima prioridad de acuerdo con una función de evaluación (lista ordenada de modo que el primer elemento de la cola es el que tiene mayor prioridad)

— Para cada estrategia de búsqueda se define una función de evaluación ($f(n)$) que devuelve un valor numérico para el nodo $\boldsymbol{n}$ tal que el nodo se inserta en la cola de prioridades en el mismo orden en el que sería expandido por la estrategia de búsqueda.

```
function GRAPH-SEARCH (problema)  return solución o fallo
    Inicializar la lista OPEN con el estado inicial del problema
    Inicializar la lista CLOSED a vacío
    do
        if OPEN está vacía then return fallo
        p←pop(lista OPEN)
        if p=estado final then return solución p
        generar hijos de p
        para cada hijo de n de p:
        aplicar f(n)
        if (n no está en CLOSED) then
            if (n no está en OPEN o (n está repetido en open
                y f(n)  es mejor que el valor del nodo en OPEN) then
                    insertar n en orden creciente de f(n) en OPEN*
        else if (f(n) es mejor que el valor del nodo repetido en CLOSED) then
            escoger entre reexpandir n (insertarlo en OPEN y eliminarlo de CLOSED)
            o descartarlo
    enddo
end

*Como estamos interesados en encontrar únicamente la primera solución,
se puede eliminar el nodo repetido en OPEN.
```

ALGORITMO TREE-SEARCH.

Los algoritmos de búsqueda se diferencian, básicamente, en la elección del siguiente nodo a expandir (**estrategia de búsqueda**).

```
function TREE-SEARCH (problema)  return solución o fallo
    Inicializar la lista OPEN con el estado inicial del problema
    do
        if (lista OPEN está vacía) then return fallo
        expandir un nodo:escoger nodo hoja y eliminarlo de la lista OPEN
        if (nodo escogido es el estado final) then return solución
        generar hijos y añadir los nodos resultantes a la lista OPEN
enddo
```

ESTADOS REPETIDOS.

Espacio de búsqueda $\neq$ Espacio de estados (el árbol de búsqueda puede contener estados repetidos y ciclos). Con el fin de evitarlos

— Incluir en el algoritmo TREE-SEARCH una estructura de datos para guardar los nodos explorados o expandidos (lista CLOSED).

— Cuando se genera un nuevo nodo, si éste se encuentra en la lista CLOSED (nodos ya expandidos) o en la lista OPEN (nodos aún no expandidos) se puede eliminar en lugar de añadirlo a la lista OPEN.

— Nuevo algoritmo: GRAPH-SEARCH, algoritmo que separa el grafo del espacio de estados en dos regiones: la región de nodos explorados/expandidos y la región de nodos no expandidos.

BÚSQUEDA A*.

El algoritmo **A*** es el más conocido de búsqueda *primero-el-mejor*.

Evalúa nodos mediante $\boldsymbol{f(n) = g(n) + f(n)}$ donde

— $\boldsymbol{g(n)}$ es el coste de alcanzar el nodo $\boldsymbol{n}$ y
— $\boldsymbol{h(n)}$ es el valor heurístico.

$f(n)$ es el **coste total estimado** de la solución óptima a través del nodo $\boldsymbol{n}$.

Los algoritmos que utilizan una función de evaluación de la forma $f(n) = g(n) + f(n)$ se conocen como algoritmos de tipo A.

La **búsqueda A*** utiliza una <u>función heurística admisible</u>.

Al utilizar una heurística admisible, la búsqueda <u>**A*** siempre devuelve la solución óptima</u>.

PROPIEDADES.

— **COMPLETA.**
→ Sí, a menos que existan infinitos nodos $\boldsymbol{n}$ tales que $f(n) < f(G)$.
— **ÓPTIMA.**
→ Sí, si se cumple la condición de **consistencia** (para la versión GRAPH-SEARCH).
› Siempre existe al menos un nodo $\boldsymbol{n}$ en OPEN que pertenece al camino óptimo de la solución.
› Si C^* es el coste de la solución óptima:
› A* expande todos los nodos con $f(n) < C^*$.
› A* podría expandir algunos nodos con $f(n) = C^*$.
› A* no expande nodos con $f(n) > C^*$.
— **COMPLEJIDAD TEMPORAL.**
→ $\boldsymbol{O(b^{C*/min_{coste_{accion}}})}$; exponencial con la longitud del camino.
→ El número de nodos expandidos es exponencial con la longitud de la solución.
— **COMPLEJIDAD ESPACIAL.**
— Mantiene todos los nodos en memoria (como todas las versiones de GRAPH-SEARCH).
→ A* normalmente se queda sin espacio mucho antes de que se agote el tiempo.
→ Por tanto, el mayor problema es el espacio, no el tiempo.

COSTE UNIFORME.

Expandir el nodo con el menor coste (menor $g(n)$).

Función de evaluación (cola de prioridades):

$$f(n) = g(n)$$

PROPIEDADES.

— **COMPLETA.**
→ Si los costes de las acciones $\geq \varepsilon$ (constante positiva → costes no negativos).
— **ÓPTIMA.**
→ Si los costes de las acciones son no negativos $g(sucesor(n)) > g(n)$.
→ Coste uniforme expande nodos en orden creciente de coste.
— **COMPLEJIDAD TEMPORAL.**
Sea $\boldsymbol{C^*}$ el coste de la solución óptima y asumimos que todas las acciones tienen un coste mínimo de ε.
→ Complejidad temporal y espacial: $O(b^{C^*/\varepsilon})$.

HEURÍSTICA.

— $h(n) = 0 \rightarrow$ coste computancional de $h(n)$ nulo. (Búsqueda lenta. Admisible).
— $h(n) = h^*(n) \rightarrow$ coste computacional de $h(n)$ alto. (Búsqueda rápida. Admisible).
— $h(n) > h^*(n) \rightarrow$ coste computacional de $h(n)$ muy alto. (Búsqueda muy rápida. No admisible).

En problemas muy complejos, merece la pena $h(n) > h^*(n)$.

ADMISIBILIDAD.

Una heurística es **admisible** si nunca sobreestima el coste para lograr el objetivo.

Formalmente

— una heurística $h(n)$ es **admisible** si $\forall n$, $h(n) \leq h^*(n)$, donde $h^*(n)$ es el coste real de alcanzar el objetivo desde el estado $\boldsymbol{n}$ y
— $h(n) \geq 0$, así que $h(G) = 0$ para cualquier objetivo $\boldsymbol{G}$.

CONSISTENCIA.

$\boldsymbol{h(n)}$ es consistente si, para cada nodo $\boldsymbol{n}$ y cada sucesor $\boldsymbol{n'}$ de $\boldsymbol{n}$ generado con una acción $\boldsymbol{a}$ se cumple $\boldsymbol{h(n) \leq h(n') + c(n, a, n')}$.