**Resumen Programación 3**

# Resolución de problemas mediante búsqueda

## Problemas de búsqueda

Un problema de búsqueda se define formalmente por sus 5 componentes:

1. **Estado inicial.**
2. **Acciones.**
3. **Modelo de transiciones. = RESULTADO(S,A)**
4. **Test objetivo**
5. **Costo de camino.**

**Formular**. Proceso de expresar un problema como un problema de búsqueda eligiendo la representación para los estados y las acciones que lo describen.

**Abstraer**. Proceso de remover de una representación detalles irrelevantes.

**Estado**. Es una representación de un instante de los elementos de un problema.

**Acción**. Modifica el estado actual.

1. **Estado inicial**. Estado del problema en el instante inicial.
2. **Acciones** devuelve el conjunto de acciones que se pueden ejecutar en estado S.

**Predicado** es una función que retorna True o False.

1. **Modelo de transiciones.** Describe el resultado de aplicar las acciones a los estados

Dado un estado S y una acción A e ACCIONES(S), la función RESULTADO(S, A) retorna el estado que resulta de aplicar A en S.

Si RESULTADO(S, A) = S’, entonces S’ es un **sucesor** de S.

**Estado objetivo**

**Objetivo**. Es un estado al que se desea llegar.

Pueden existir múltiples objetivos en un problema.

**Espacio de estados**.

El estado inicial, las acciones y el modelo de transiciones definen implícitamente el **espacio de estados** del problema. Representa el conjunto de estados alcanzables desde el estado inicial mediante cualquier secuencia de acciones.

Cuando es finito, se puede representar con un **grafo dirigido** (o **digrafo**) donde los **nodos** son estados y los arcos son acciones.

Un **camino** en el espacio de estados es una secuencia de estados conectados mediante una secuencia de acciones.

1. **Test Objetivo.**

Dado un estado S, el predicado TEST-OBJETIVO(S) determina si S es un objetivo.

Puede darse mediante una **enumeración explícita** de todos los objetivos o mediante una **propiedad**.

**Enumeración explícita** sería en el ejemplo del cruce de río donde:



Propiedad sería en el ejemplo de Ajedrez donde:



1. **Costo de camino**

Todo camino del espacio de estados tiene un costo numérico (no negativo) asociado, denominado **costo de camino**.

* Dado un camino P, la función COSTO-CAMINO(P) retorna el costo de camino de P.

En las primeras unidades, asumimos que el costo de camino es igual a la suma de los costos individuales de cada acción del camino.

Por lo tanto, si P = <S0 , A0 , …, An-1 , Sn >



* Dado un estado S y una acción A e ACCIONES(S), la función COSTO-INDIVIDUAL(S,A) retorna el costo individual de aplicar A en S.

En el ejemplo del río el costo individual de cada acción es 1. Y el costo de camino es exactamente la cantidad de acciones realizadas

En general, un problema puede admitir múltiples formulaciones, en base a la representación de estado y acción elegida.

## Solución

Dado un problema, una **solución** es un **camino** en el espacio de estados desde el **estado inicial** a un **objetivo**.

Una **solución óptima** tiene el **menor costo de camino** entre todas las soluciones.

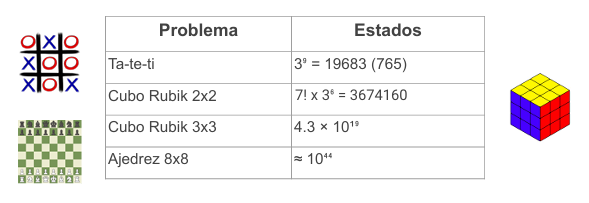
## Algoritmos de búsqueda

Formulado nuestro problema, lo siguiente es resolverlo, es decir, encontrar una solución, preferentemente óptima.

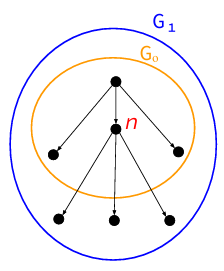
Este proceso se conoce como búsqueda y a los algoritmos específicos para esta tarea, algoritmos de búsqueda.

Obviamente en los problemas chicos se puede resolver a mano. Pero esto no escala con el tamaño del problema y no será posible en general.

Problemas más interesantes suelen tener miles o millones de estados.

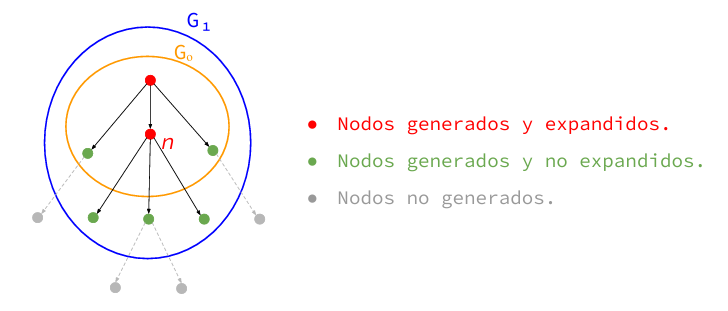


### Búsqueda de grafo.

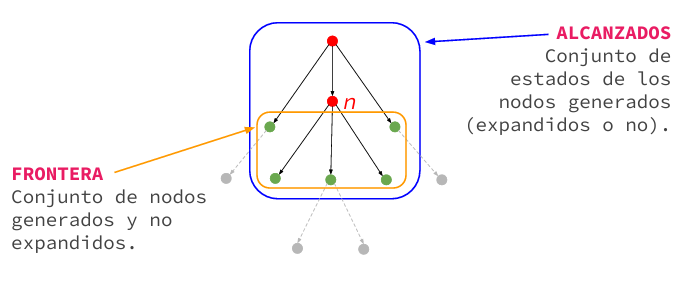


* En general, **no es posible almacenar el grafo de espacio de estados por completo en memoria**.
* La idea es **buscar soluciones mientras lo construímos**. Con suerte, encontramos una solución antes de construirlo por completo.
* Informalmente, comenzamos buscando en un subgrafo más chico G0.
* Si no encontramos solución en G0, elegimos un nodo “n” de G0 y lo **expandimos,** es decir, agregamos a G0 un nodo por cada nuevo vecino de “n”.
* Obtenemos un grafo G1 desde donde continuamos la búsqueda.

### Clasificación de nodos al buscar



### Alcanzados y Frontera



### Búsqueda de grafo

1. **Inicialización**: construir un nodo n₀ con el estado inicial s₀, poner en **ALCANZADOS** a s₀ y en **FRONTERA** a n₀
2. Iteración: mientras FRONTERA no esté vacía:
   1. Selección: quitar algún nodo n de FRONTERA.
   2. Test objetivo: si n tiene un estado objetivo, retornar la solución.
   3. Expansión: de lo contrario, expandir n y actualizar FRONTERA y ALCANZADOS con sus vecinos no alcanzados.

### Eficiencia de la búsqueda

Un 23% de ahorro puede parecer insuficiente, pero en problemas con grandes espacios de estados, el ahorro puede ser muchísimo mayor.

El ahorro depende de:

* La estructura del grafo de espacio de estados.
* El orden en que se eligen los nodos de la frontera.

En general, en problemas con grandes espacios de estados, elegir de manera inteligente los nodos de la frontera puede hacer la diferencia entre poder o no resolverlo

### Árbol de búsqueda

Las posibles secuencias de acciones desde el estado inicial forman un **árbol de búsqueda.**

* Cada **nodo** contiene algún estado del espacio de estados. En particular, la **raíz** contiene el estado inicial.
* De cada nodo (**padre**) y por cada acción, hay una **rama** que lo conecta con un nodo (**hijo**) con el estado sucesor..
* Una **hoja** es un nodo sin hijos (estados sin sucesores).

El árbol de búsqueda describe todos los posibles caminos desde el estado inicial en el espacio de estados.

El árbol de búsqueda puede tener nodos con estados repetidos.

Si en el árbol de búsqueda se podan nodos con estados repetidos, entonces cada estado aparece a lo sumo una vez.

Esto significa que podemos pensar al árbol de búsqueda dentro del grafo de espacio de estados.

Justamente este tipo de árbol de búsqueda es el que construye el algoritmo de búsqueda de grafo.

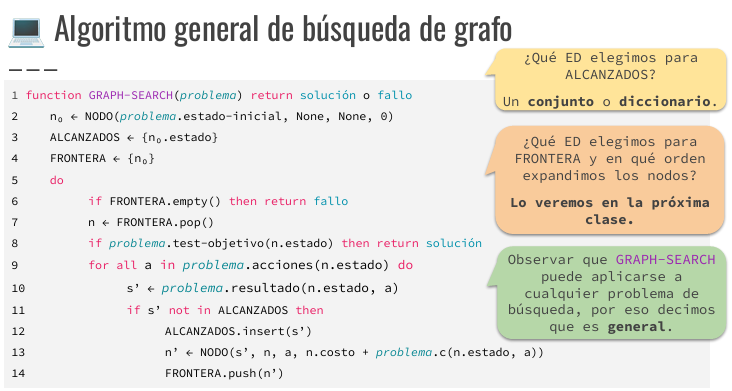
**Una implementación de esto puedes ser:**

Un nodo n se puede representar con una estructura con 4 o 5 atributos:

* **n.estado**: estado que representa a n.
* **n.padre**: nodo del árbol que generó a n.
* **n.acción**: acción que aplicada al padre para generar a n.
* **n.costo**: costo de camino de la raíz a n.
* **n.profundidad** (opcional): nivel del árbol en el que se encuentra n.

Este atributo que apunta al padre sirve para poder recuperar fácilmente una solución. Una vez encontrado un nodo objetivo, retrocedemos por los mismos hasta la raíz.

En Prog2 generalmente, usábamos n.hijo.



El consumo de memoria de **GRAPH-SEARCH** depende principalmente de **el máximo número de estados en alcanzados**

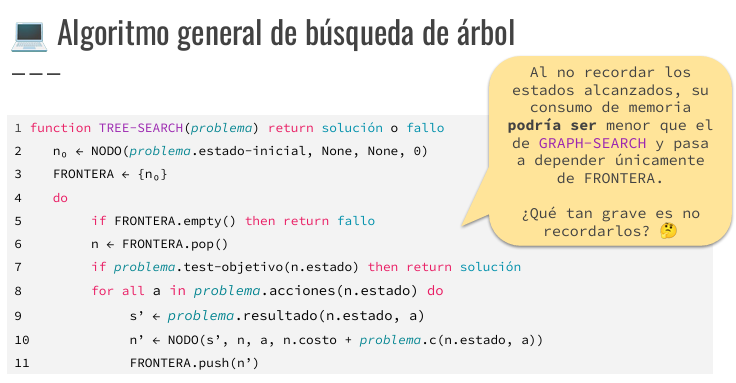
### Eficiencia de GRAPH-SEARCH

* El consumo de memoria de **GRAPH-SEARCH** puede volverse prohibitivo
  + En el peor de los casos, **ALCANZADOS podría contener tantos estados como estados alcanzables tenga el problema**.

### Búsqueda de árbol

La **búsqueda de árbol** es similar a la búsqueda de grafo, salvo que no se mantiene en memoria un conjunto con los estados alcanzados previamente.

Por supuesto, el árbol de búsqueda que se construye podría contener nodos con estados repetidos.



Este tipo de búsqueda tiene el problema de tener la posibilidad de seguir caminos cíclicos.

### Caminos cíclicos

Un camino cíclico en el espacio de estados es un camino que pasa al menos dos veces por un mismo nodo.

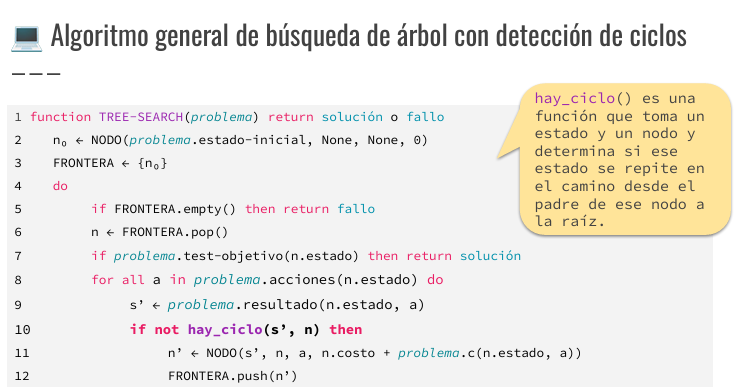
En particular, un problema con acciones reversibles siempre tendrá caminos cíclicos.

En estos casos, el árbol de búsqueda es infinito aún cuando el espacio de estados es finito. En consecuencia, TREE-SEARCH podría no terminar.

### Detección de caminos cíclicos

Antes de agregar un nuevo nodo a FRONTERA, controlamos que no repita estados en el camino de la raíz a su padre.

Dado que cada nodo almacena a su padre, este control es fácil de agregar y no es demasiado costoso.



Ahora el siguiente problema son los caminos redundantes, es decir, caminos que tienen estados diferentes, pero terminan en el mismo estado.

### Caminos redundantes

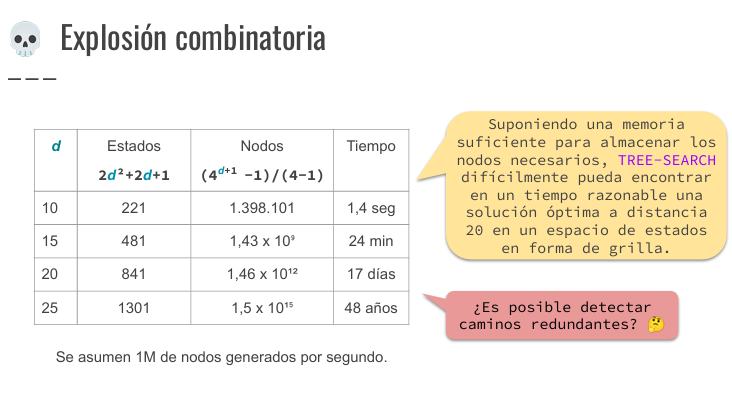
Dos caminos distintos en el grafo de espacio de estados que inician y finalizan en el mismo nodo se llaman **caminos redundantes**.

**La existencia de caminos redundantes permite que haya nodos con estados repetidos en el árbol de búsqueda construido por TREE-SEARCH, incluso en la versión con detección de ciclos.**

Sea un problema con un grafo de espacio de estados en forma de grilla.

El número de estados a distancia a lo sumo d de S0 responde a la fórmula: 2d^2 + 2d +1

Debido a los caminos redundantes, el número de nodos del árbol de búsqueda hasta el nivel d (sin detectar ciclos) responde a la fórmula: 1 + 4^1 + … + 4^d = (4^d+1 - 1)/(4-1)



### Detección de caminos redundantes

La forma de evitar caminos redundantes es recordar los estados alcanzados en el pasado, es decir, volviendo a la búsqueda de grafo.

“Los algoritmos que olvidan su historia están condenados a repetirla”.

### GRAPH-SEARCH VS TREE-SEARCH

Decidir si usar uno u otro no es sencillo.

* Si el espacio de estados tiene pocos caminos redundantes, o ninguno, entonces TREE-SEARCH será más eficiente.
* Por el contrario, si el espacio de estados tiene muchos caminos redundantes, la única alternativa será GRAPH-SEARCH.

### Criterios de evaluación para algoritmos de búsqueda

1. **Completitud**: si hay solución, ¿encuentra al menos una?
2. **Optimalidad**: si hay solución, ¿encuentra la óptima?
3. **Tiempo**: ¿Cuántos nodos genera?
4. **Memoria**: ¿Cuánta memoria usa?

# Estrategias de búsqueda no informadas

## Búsqueda primero en anchura

Breadth-First Search (BFS)

Los nodos del árbol de búsqueda se expanden por niveles.

Primero se expande la raíz, luego los hijos de la raíz, luego los hijos de los hijos de la raíz, y así hasta encontrar un estado objetivo.

### Criterio de parada

En la estrategia BFS, es conveniente ejecutar el test objetivo antes de agregar un nodo a la frontera. En el mejor de los casos, evitamos expandir un nivel del árbol.

### Definiciones de teoría de grafos

Dado k e N0 , un árbol es k-ario si cada nodo tiene a lo sumo k hijos.

Dado un árbol T, el **factor de ramificación** de t es el mínimo de b e N0 tal que T es b-ario.

Un árbol k-ario está **lleno** si todos los nodos tienen 0 o k hijos.

Un árbol k-ario está **completo** si está lleno y todas las hojas están en el mismo nivel.

### Número de nodos

Dado un árbol b-ario completo.

* Número de nodos en el nivel d: b^d
* Número de nodos hasta el nivel d: (b^d+1 - 1) / (b-1)

Esto reafirma que en BFS es conveniente aplicar el test objetivo antes de agregar a FRONTERA.

Por ejemplo si suponemos un problema cuyo árbol de búsqueda tiene un factor de ramificación b = 5 y el menor nivel con un objetivo es d = 6, entonces:

Si el test objetivo se aplica antes de agregar a FRONTERA: (5^6+1 -1)/(5-1) = 19531

Si el test objetivo se aplica después de agregar a FRONTERA: (5^7+1 -1)/(5-1) = 97656

b veces menos nodos.

### Implementación de BFS

El nodo que se elige en la frontera es el de menor profundidad.

Se utiliza TAD Cola para poder lograr esto



### Completitud de TREE-BFS

**Siempre encuentra una solución con profundidad mínima** (aunque no se detecten caminos cíclicos)... Salvo que el factor de ramificación sea infinito.

### Optimalidad de TREE-BFS

Depende de los costos.

Para costos individuales unitarios, siempre encuentra una óptima. En estos casos, la profundidad de un nodo coincide con su costo, luego la solución de menor profundidad es también la de menor costo.

Para otros costos individuales puede no ser óptimo.

### Consumo de tiempo de TREE-BFS

Sea b el factor de ramificación y d el menor nivel con una solución.

En el peor caso, se generan todos los nodos hasta el nivel d, luego el número de nodos generados está acotado por: (b^d+1 -1)/(b-1) ≤ b^d+1

Puede tardar mucho tiempo.

Consumo de memoria de TREE-BFS

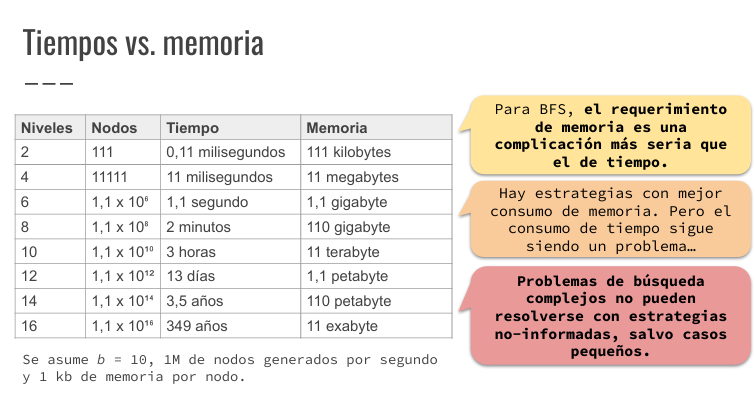
El máximo número de nodos en la frontera está acotado por el máximo número de nodos en el nivel d: b^d

Sin embargo, sus ancestros no pueden liberarse de la memoria, ya que son necesarios para reconstruir la solución. En el peor caso, se deben almacenar en memoria todos los nodos hasta el nivel d, quedando acotado por: (b^d+1 -1)/(b-1) ≤ b^d+1

### Performance de TREE-BFS

* Completitud √
* Optimalidad.
  + Cuando las acciones tienen el mismo costo individual. √
  + En otros casos. X
* Tiempo. Se genera a lo sumo b^d+1 nodos.
* Memoria. Se mantienen a lo sumo b^d+1 nodos en memoria.

Muy caro en tiempo y memoria para problemas complejos.

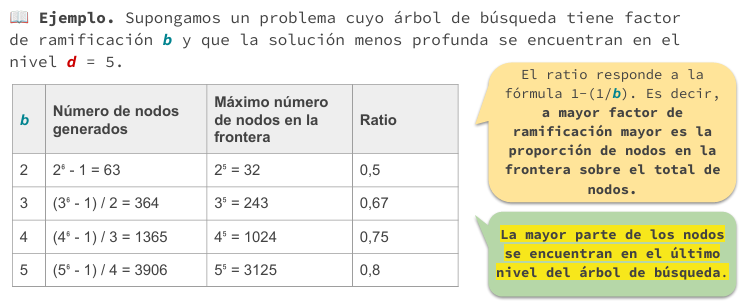


### Consumo de memoria de la frontera

¿Y si no estamos interesados en encontrar una solución sino su costo?

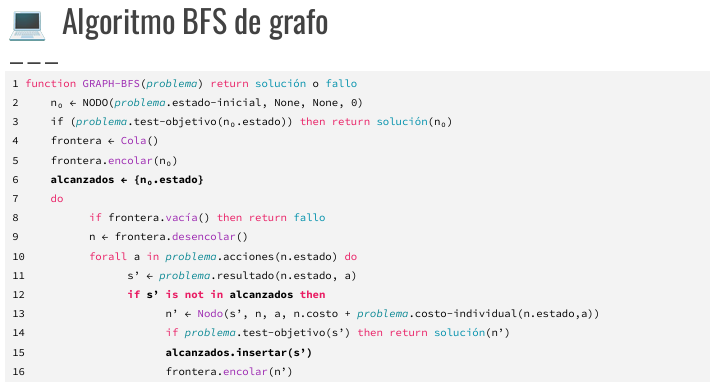
En ese caso, podríamos mantener en memoria únicamente a los nodos de la frontera y no a todos sus ancestros.

Sin embargo, la mejora obtenida no sería significativa, como veremos en el siguiente ejemplo…



### Algoritmo BFS de grafo

* Durante la ejecución, se almacenan en un conjunto los estados alcanzados.
* Al expandir un nodo, se descartan los nodos con estados ya alcanzados. Para los restantes, se marcan sus estados como alcanzados y se encolan a la frontera.



### Performance de GRAPH-BFS

* Teniendo la ventaja de que cuando un nodo ya había sido alcanzado previamente. El nodo descartado conduce a un camino redundante con costo mayor o igual.
* Por lo tanto, es óptimo y completo para costos individuales unitarios.
* El consumo de tiempo y memoria depende del problema en sí, pero en problemas con muchos caminos redundantes, el consumo de tiempo y memoria puede ser significativamente menor. Aunque en el peor caso, se podrían generar tantos nodos como estados alcanzables tenga el problema.
* En cambio, si no hay caminos redundantes, el consumo de tiempo es similar y el de memoria es mayor pero no de forma significativa. (comparado con el TREE-BFS)

## Búsqueda primero en profundidad

Depth-First Search (DFS)

Los nodos del árbol de búsqueda se expanden por profundidad.

Siempre se expande el nodo más profundo de la frontera.

La búsqueda desciende rápidamente a una hoja. Para continuar, se retrocede a su ancestro más profundo con hijos aún no expandidos y se elige uno de ellos.

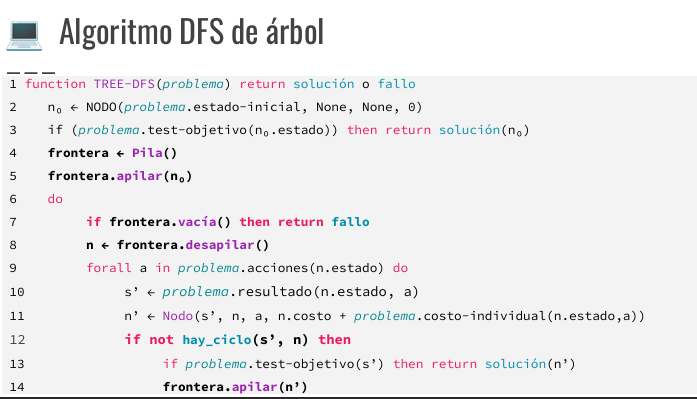
### Criterio de parada

Al igual que en BFS, en la estrategia DFS es conveniente ejecutar el test objetivo antes de agregar un nuevo nodo a la frontera. Aunque en este caso, la ganancia es más dificil de medir.

### Implementación de DFS

El nodo que se elige de la frontera es el de mayor profundidad.

Esto se puede lograr mediante el TAD Pila, de manera que siempre el nodo más profundo es el último en ser agregado a la Pila y es el primero en ser desapilado.



### Completitud de TREE-DFS

Si el espacio de estados es infinito o finito pero no se detectan caminos cíclicos, entonces el árbol de búsqueda es infinito y DFS podría seguir un camino descendente infinito desde la raíz, sin nunca pasar por un nodo con un estado objetivo.

En cambio, si el espacio de estados es finito y se detectan caminos cíclicos, entonces el árbol de búsqueda es finito (con a lo sumo m niveles) y DFS siempre encuentra una solución.

### Optimalidad y consumo de tiempo de TREE-DFS

No es óptimo, ya que al ir en profundidad es capaz de encontrar una solución, pero esta puede estar en un nivel mayor que d, siendo d el menor nivel de una solución.

En torno al consumo de tiempo, en el peor caso, se generan todos los nodos del árbol de búsqueda. Si hay m niveles y el factor de ramificación es b, entonces el número de nodos generados está acotado por: (b^m+1 - 1) / (b-1) <= b^m+1

### Consumo de memoria de TREE-DFS

Observar que es necesario mantener en memoria al nodo actual n, a sus ancestros y a la frontera (la cual contiene a los hermanos no expandidos de n y de sus ancestros)

Si hay m niveles y el factor de ramificación es b, entonces el máximo número de nodos en memoria está acotado por: b\*m

### Performance de TREE-DFS

* Completitud.
  + Si el espacio de estados es infinito o es finito pero no se detectan ciclos. X
  + Si el espacio de estados es finito y se detectan ciclos. √
* Optimalidad. X
* Tiempo. Se genera a lo sumo b^m+1 nodos. Si se detectan ciclos, el consumo de tiempo es proporcional a m\*b^m+1.
* Memoria. Se mantienen a lo sumo b\*m nodos en memoria.

El consumo de memoria es lineal, el consumo de tiempo es demasiado grande dependiendo el problema.

### TREE-BFS vs. TREE-DFS

Suponiendo b = 10 y 1kb por nodo, en el nivel d = m = 16

BFS consume: 11 exabytes

y DFS consume: 160 kilobytes

es decir, 6,9 x 10^12 veces menos de memoria.

DFS requiere mucha menos memoria, pero se pierde la garantía de optimalidad.

### Algoritmo DFS de grafo

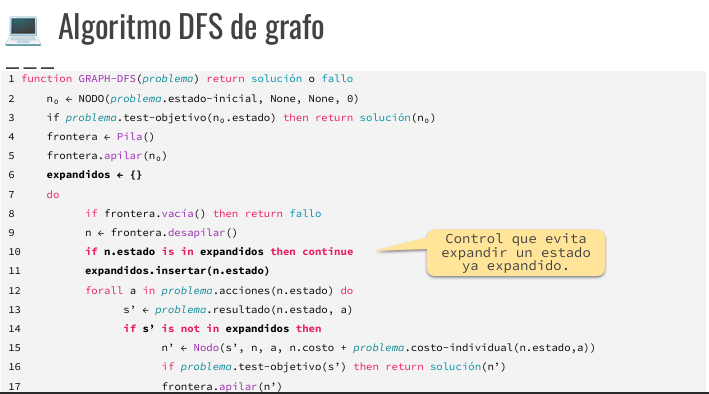
Suponer que al expandir un nodo **m**, se genera un hijo **n** con un estado **s** tal que en la frontera ya hay un nodo **n’** con **s** (generado previamente).

Manteniendo un conjunto de estados alcanzados, el nodo **n** sería descartado (pues **s** ya fue alcanzado al generar a **n’**), cuando lo correcto sería descartar a **n’** pues tiene menor profundidad

En lugar de mantener un conjunto de estados alcanzados, se mantiene un conjunto de estados expandidos (provenientes de nodos ya sacados de la frontera).

Ahora, en la frontera pueden aparecer **nodos con estados repetidos**, pero solo se permite expandir el de mayor profundidad.

Se agrega un control que evita expandir un nodo con un estado ya expandido.



### Performance de GRAPH-DFS

* Al igual que **TREE-BFS**, es **completo** en espacios de estados finitos y es **no óptimo**.
* El **consumo de tiempo** depende de la cantidad de caminos redundantes del problema (algo similar ocurría con **GRAPH-BFS**).
* El consumo de memoria aumenta. Pasa de ser proporcional al largo del camino más profundo a ser proporcional al número de estados alcanzables.

Dado que GRAPH-BFS tiene el mismo requerimiento de memoria y además es óptimo (para costos individuales unitarios), se prefiere a **GRAPH-BFS** como algoritmo de búsqueda de grafo.

## Búsqueda de costo uniforme

Uniform-Cost Search (UCS)

UCS es una **generalización** de BFS que extiende la optimalidad para problemas con **cualquier función de costo individual**.

En lugar de expandir el nodo de la frontera con la menor profundidad, se expande siempre el de **menor costo de camino.**

En particular, cuando los costos individuales son unitarios, el costo de camino coincide con la profundidad y la estrategia es exactamente BFS.

### Propiedades

Cuando un estado (objetivo o no) es alcanzado por primera vez, no garantiza un camino de costo mínimo a ese estado.

Cuando se extrae un nodo de la frontera, entonces garantiza un camino de costo mínimo a ese estado.

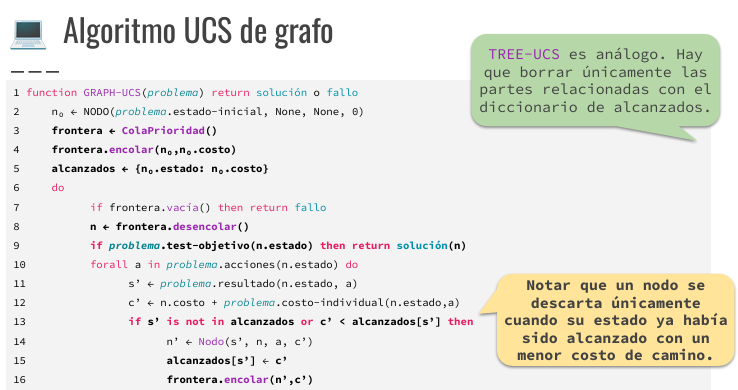
Los demás nodos de la frontera tienen un costo de camino que es mayor o igual (de lo contrario hubiesen sido extraídos anteriormente) y no puede conducir a caminos redundantes de menor costo (pues los costos individuales nunca son negativos).

### Criterio de parada

En la estrategia UCS, es conveniente ejecutar el test objetivo después de sacar un nodo de la frontera ya que garantiza optimalidad para cualquier función de costo individual.

### Implementación de UCS

El nodo que se elige de la frontera es el de menor costo de camino.

Esto se logra mediante un TAD Cola de prioridad.

### Performance de TREE-UCS y GRAPH-UCS

* Completitud √.
* Optimalidad √. **Para cualquier función de costo individual**
* Tiempo y memoria.
  + El consumo de TREE-UCS no se puede caracterizar fácilmente en términos de b (factor de ramificación) y d (profundidad mínima de una solución). La cantidad de nodos generados depende también del **menor costo individual de una acción**. No lo vamos a estudiar…

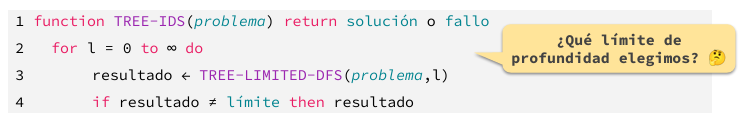
## Búsqueda de profundización iterativa

Se introduce un límite de profundidad “L”.

En cada iteración, se aplica DFS sobre el árbol de búsqueda **limitado a esa profundidad**.

Es decir, los nodos del árbol de búsqueda en el nivel “L” se tratan como si fuesen **hojas**.

Comenzando con L = 0, el límite se **incrementa** gradualmente en 1 hasta encontrar una solución.



TREE-LIMITED-DFS es muy similar a TREE-DFS.

Lo único que TREE-LIMITED-DFS puede devolver es una solución, fallo o un valor límite que indica que para el límite de profundidad actual no se encontró solución y se precisa aumentarlo.



### Performance de TREE-IDS

* Completitud ✅. (si el límite de profundidad es suficientemente grande). Siempre se encuentra solución al llegar al límite de profundidad L = d. Este mismo límite impide seguir caminos descendentes infinitos, ni siquiera generados por caminos cíclicos.
* Es óptimo ✅ en problemas de costos individuales unitarios (igual que TREE-BFS).

### Consumo de tiempo

De manera general:

* La raíz se genera d+1 veces.
* Los nodos del nivel 1 se genera d veces.
* Los nodos del nivel 2 se generan d-1 veces.
* Los nodos del nivel d se general a lo sumo 1 vez.

Luego, el número de nodos generados es a lo sumo:

(d+1)\**b^0 + d*\*b^1 + (d-1) \* b^2 + … + 1\* b^d

El consumo de tiempo de TREE-IDS puede parecer mucho mayor que el de TREE-DFS. Pero la diferencia al final no es significativa, pues la mayoría de los nodos del árbol de búsqueda se encuentran en el último nivel, y es menos significativa a mayor factor de ramificación.

### Consumo de memoria de TREE-IDS

El consumo de memoria de TREE-IDS depende exclusivamente del consumo de memoria de TREE-LIMITED-DFS.

Dado que TREE-LIMITED-DFS es muy similar a TREE-DFS, ambos algoritmos tienen el mismo consumo de memoria, es decir, b \* d

### Performance de TREE-IDS

* Completitud ✅
* Optimalidad
  + ✅ Cuando las acciones tienen el mismo costo individual.
  + X en otros casos.
* Tiempo. se generan a lo sumo b^d+1 nodos.
* Memoria. Se mantienen a lo sumo b \* d nodos en memoria.

Es decir, combina las ventajas de TREE-BFS y TREE-DFS

### Algoritmo IDS de grafo

* GRAPH-LIMITED-DFS mantiene en memoria el conjunto de estados ya expandidos, de la misma forma que GRAPH-DFS
* Es decir, su consumo de memoria pasa a depender del tamaño del conjunto de expandidos (que puede ser exponencial en la altura del árbol).
* No tiene ninguna ventaja respecto a GRAPH-BFS

# Estrategias de búsqueda informadas

* Saben si un nodo tiene un estado más prometedor que otro y lo expanden antes.
* Además de la definición del problema de búsqueda, requiere de conocimiento específico del problema.
* Incorporan una función heurística h donde, para cada nodo n, h(n): estimación del menor costo de camino desde el estado contenido en n a un estado objetivo.
* Si n es un nodo con un estado objetivo, entonces h(n) = 0.

## Búsqueda avara primero el mejor

Greedy Best-First Search (GBFS)

Intenta encontrar rápidamente una solución.

Los nodos de la frontera se expanden según su cercanía a un nodo objetivo.

Dada una función heurística h, se expande siempre el nodo n de la frontera con el menor valor heurístico.

### Observaciones

La búsqueda de árbol con estrategia GBFS:

1. Puede seguir caminos cíclicos. Es necesario detectarlos para garantizar **completitud** en espacios de estados finitos.
2. Tiende a seguir caminos en profundidad. Pero a diferencia de DFS, puede **retroceder** antes de llegar a una hoja si el camino se empieza a alejar del objetivo.
3. No tiene garantía de **optimalidad**.

### Implementación

El nodo que elegimos de la frontera para expandir es el de menor costo h(n)

Al igual que en UCS usando el TAD cola de prioridad para la frontera pero ordenando los nodos por la función heurística.



Performance de TREE-GBFS y GRAPH-GBFS

* Completitud
  + X si el espacio de estados es infinito o es finito pero no se detectan ciclos.
  + ✅ si el espacio de estados es finito y se detectan ciclos.
* Optimalidad. X
* En el peor caso, la heurística podría ser **h(n) = 0 para todo n**, es decir, todos los nodos de la frontera tienen la misma prioridad y la búsqueda se vuelve no informada. Por lo tanto, podría tener el mismo **consumo de tiempo** **que** **TREE-DFS** y el mismo **consumo de memoria que TREE-BFS**, ambos exponenciales. **Sin embargo, con una heurística apropiada, la cantidad de nodos generados puede reducirse drásticamente.**

### Entre UCS y GBFS

* UCS garantiza optimalidad, pero genera todos los nodos con costo de camino menor al de una solución óptima.
* GBFS puede reducir drásticamente la cantidad de nodos generados con una función heurística apropiada, pero no garantiza optimalidad.

## Búsqueda A\*

A\* Search

Combina las estrategias UCS y GBFS.

Elige el nodo a expandir según una combinación de su costo de camino y su valor heurístico.

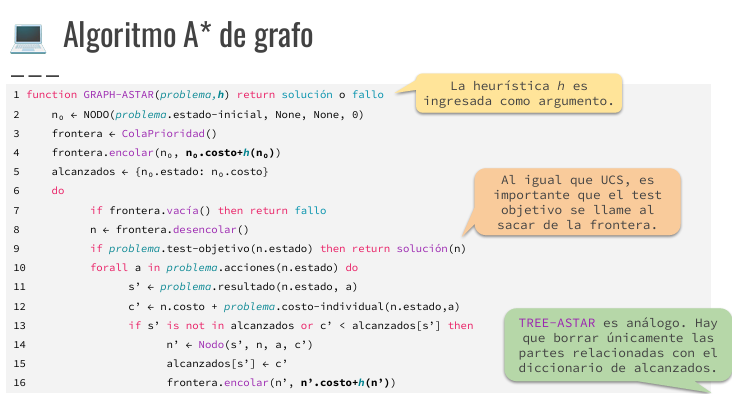
Se define una función de evaluación f tal que, dado un n, **f(n) = n.costo + h(n)** es decir, f(n) es el **costo estimado de la solución menos costosa** que pasa por n.

El nodo n que se extrae de la frontera es siempre el de **menor valor de evaluación** f(n)

### Implementación

El nodo n con la menor evaluación f(n) va a ser el próximo nodo a expandir de la frontera.

Al igual que en UCS y GBFS, usando el TAD cola de prioridad para la frontera pero ordenando los nodos por la función de evaluación f.



NOTA: el costo que se guarda sigue siendo el mismo, en lo único que utilizamos la heurística es para evaluar qué camino tomar.

### Completitud y optimalidad

**La completitud y optimalidad dependen de la heurística.**

Definición. Una heurística h es **admisible** si, para todo nodo n, h(n) nunca sobrestima el mínimo costo de camino desde el estado contenido en n a un estado objetivo.

Definición. Una heurística h es **consistente** si, para todo n e hijo n’ generado por una acción a, se cumple:

h(n) <= COSTO-INDIVIDUAL(n.estado, a) + h(n’)

**Propiedad**. Toda heurística consistente es admisible.

**Teorema**.

* TREE-ASTAR garantiza completitud y optimalidad si la heurística h es **admisible**.
* GRAPH-ASTAR garantiza completitud y optimalidad si la heurística h es **consistente**.

Demostración. no la vamos a ver

### Consumo de tiempo y memoria

El consumo de tiempo y memoria de TREE-ASTAR es difícil de medir porque depende de la heurística. Por ejemplo, si la heurística es h(n) = 0 para todo n, entonces es equivalente a TREE-UCS, y por lo tanto tiene el mismo consumo de tiempo y memoria.

A su vez, en problemas con muchos caminos redundantes, GRAPH-ASTAR puede reducir drásticamente el consumo de tiempo y memoria respecto a TREE-ASTAR.

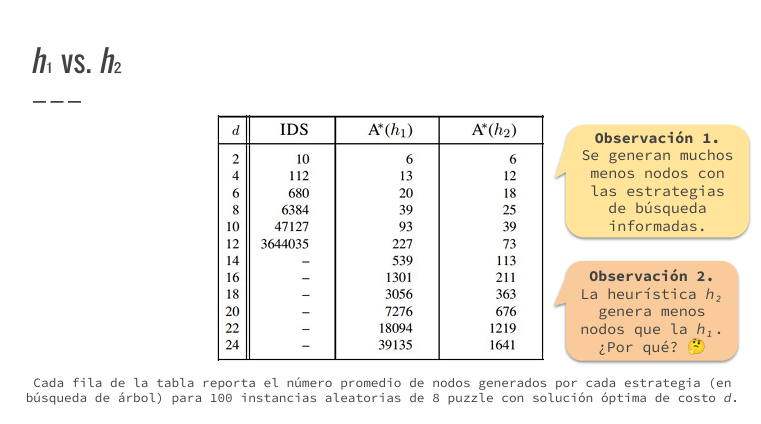
### Cómo construir heurísticas

Ejemplos de heurísticas para el 8-puzzle

h1: Número de fichas (numéricas) mal ubicadas.

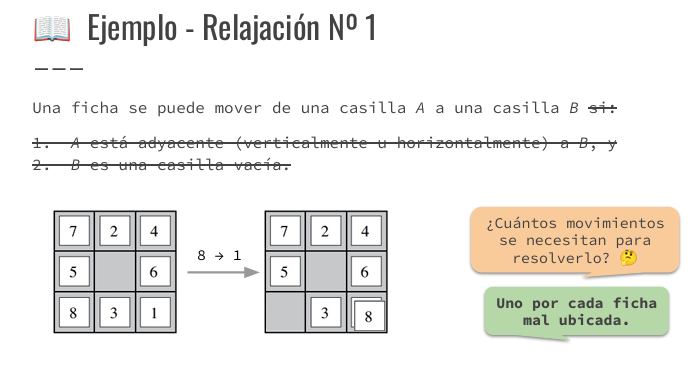
h2: suma de la **distancia de Manhattan** de cada ficha (numérica) a su posición correcta.

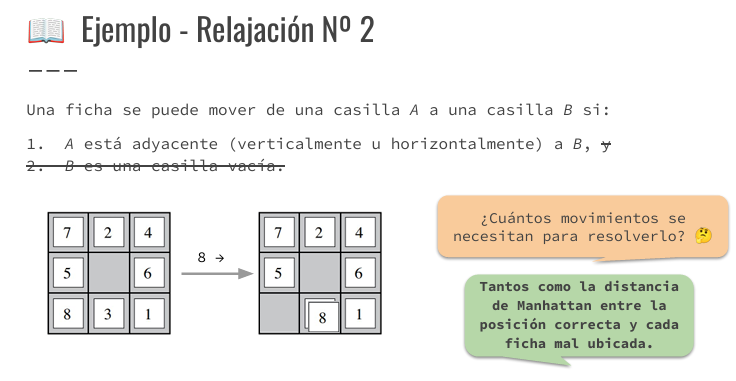
### h1 vs h2



* Se puede probar que ambas son **admisibles**.
* ¿h2 es siempre mejor que h1? **Si**.
* Definición. Sean dos heurísticas h1 y h2. Se dice que h2 **domina** a h1 si, para todo n, h2(n) >= h1(n)
* Teorema. Sean dos heurísticas h1 y h2 tales que h2 domina a h1, **A\* con h2 nunca expande más nodos que A\* con h1**.

## Problemas relajados

Un problema relajado se obtiene quitando restricciones a las acciones de un problema.



### Propiedades

Sea P’ un problema relajado de un problema P.

1. El grafo de espacio de estados de P es un subgrafo del de P’.
2. El costo de una solución óptima en P’ es una cota inferior del costo de una solución óptima en P.
3. Si P’ se **resuelve sin búsqueda** podemos derivar una heurística para P.
   1. Relajación N°1 -> Heurística h1.
   2. Relajación N°2 -> Heurística h2.

### Admisibilidad y consistencia

Proposición. Sean P’ un problema relajado de un problema P y h la heurística proveniente de P’, entonces h es **admisible** y **consistente** para P.

## Subproblema

Un subproblema consiste en resolver una parte de un problema más grande.

En general, se obtiene quitando restricciones del objetivo de un problema

Un ejemplo sería que en vez de tener que ubicar las 8 fichas del 8-puzzle, tuviéramos que colocar solo 4 en su posición.

### Propiedades

Sea un subproblema P’ de un problema P.

1. Una solución óptima de P’ puede extenderse a una óptima de P.
2. El costo de una solución óptima de P’ es una cota inferior del costo de una solución óptima de P.
3. Si P’ se resuelve rápidamente con una búsqueda, podemos derivar una heurística para P.

### Espacio de estados

En el 8-puzzle el número de estados alcanzables es: 9!/2 = 181440

En el subproblema 1-2-3-4, representando a las fichas 5-6-7-8 con \*, el número de estados alcanzables es: 15120

El espacio de estados es 12 veces más pequeño.

### Modelo de bases de datos - Pattern database

Un **modelo de bases de datos** almacena en memoria el costo de una solución óptima **para cualquier estado inicial del subproblema**

Luego, para computar h(n) basta con buscar en la base de datos el costo correspondiente.

### Debilidades

La heurística de la base de datos 1-2-3-4 es un poco débil, pues no considera las demás fichas…

No podemos sumar las heurísticas, ya que podríamos sobrestimar

### Combinar heurísticas

* Las soluciones de los subproblemas seguro comparten acciones, las cuales se contarían más de una vez.
* Para sumarlas, no se deben contar los movimientos que desplacen a una ficha \*, es decir, esas acciones deben considerarse con costo individual nulo.

Esto permite que nos dé una estimación para el problema original, no siempre será exacto, pero de seguro no sobrestima.

### Modelo de bases de datos disjuntas

* Las heurísticas de dos bases de datos se pueden sumar siempre que sean **disjuntas.**
* Esto se conoce como **modelo de bases de datos disjuntas.**
* Permiten resolver el 15-puzzle en milisegundos.

## Heurística compuesta

Sean h1,...,hm heurísticas para un problema y tales que ninguna domina la otra.

Se define la **heurística compuesta**: h(n) = max{h1(n),...,hm(n)}

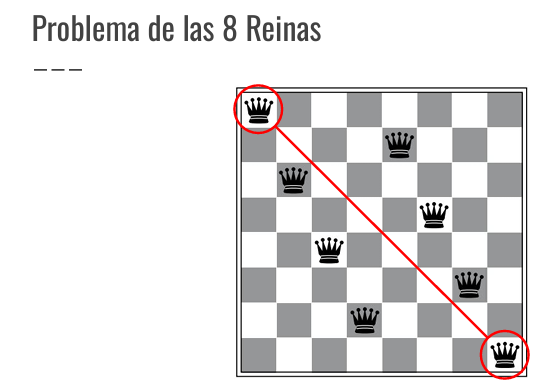
### Admisibilidad y consistencia

Proposiciones.

* Si h1,...,hm son admisibles, entonces h es **admisible**
* Si h1,...,hm son consistentes, entonces h es **consistente.**
* h **domina** a h1,..,hm.

# Búsqueda local

Estamos interesados en encontrar un estado objetivo en el espacio de estados. Sin importar el camino en el espacio de estados desde el estado inicial a ese estado objetivo



¿Es posible resolver este problema?

En total, hay 92 soluciones. Sin contar soluciones simétricas, hay 12 soluciones.

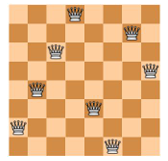
Hay un problema con la forma en que veníamos planteando los problemas que es que en problemas grandes el número de estados alcanzables se vuelve muy grande o el número de nodos en memoría sería gigantesco.

### Deducción de soluciones

El problema de las reinas tiene una particularidad…

A partir de un estado, se puede deducir fácilmente la secuencia de acciones que lo generan.

Para el estado de la izquierda, y considerando la formulación incremental (II)(la que deja solamente colocar reinas de izq a derecha, y no deja que se ataquen), se tiene la siguiente secuencia de acciones: 7 > 5 > 3 > 1 > 6 > 8 > 2 > 4



En muchos problemas, si conocemos un estado objetivo, es trivial encontrar una solución (es decir, un camino en el espacio de estados desde el estado inicial a él).

Esto permite pensar en nuevos algoritmos de búsqueda…

* No necesitamos un puntero al padre del nodo.
* No necesitamos mantener en memoria a los ancestros.
* Pero la ganancia puede no ser suficiente (ya hemos hablado del tamaño de la frontera) (capaz ni es necesario mantenerla)

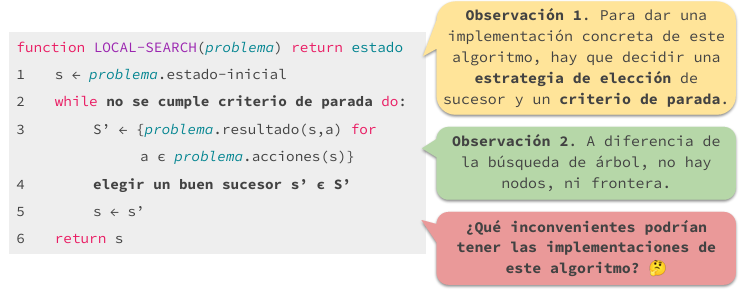
## Búsqueda local

Local search

Una **búsqueda local** mantiene unos **pocos** estados en memoria. En cada iteración, **actualiza** el estado actual por alguno de sus sucesores, **eligiéndolo** con algún criterio y **deteniéndose** con algún criterio.

Hay una **amplia** familia de algoritmos que se pueden clasificar como búsqueda local.

**Aplicación**. Problemas donde lo importante es encontrar un estado objetivo (y no el camino al mismo)



### Características generales

* Muy bajo consumo de **memoria**.
* Pueden encontrar soluciones **razonables** en espacios de estados muy grandes (o infinitos)
* Se **atascan** en estados terminales (sin sucesores).
* Podrían **ciclar**.
* No garantizan **completitud** ni **optimalidad.**

### Formulación incremental vs Formulación de estados completa

Las formulaciones incrementales son poco útiles con búsqueda local, dado que **es imposible progresar cuando se llega a un estado terminal que no es objetivo**.

Con búsqueda local, se usan típicamente formulaciones de estados completa, donde cada estado ya tiene ubicadas todas las componentes del problema y siempre es posible aplicar acciones que muevan o intercambien las componentes ya ubicadas.

### Costos de camino

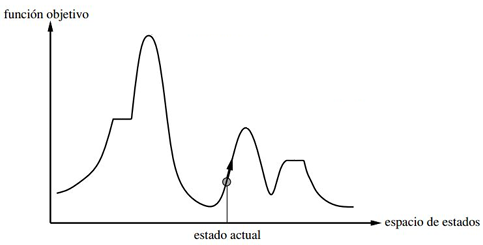
En una búsqueda local, los costos de camino son **irrelevantes**, dado que **se sigue un único camino y no se lleva ningún registro de las alternativas que quedaron sin explorar**.

### Funciones objetivo

* Los algoritmos de búsqueda local, se guían por medio de una **función objetivo**.
* Una función objetivo f, asigna a cada estado s, un valor numérico f(s), conocido como valor objetivo (de s por f).
* Una función objetivo tiene que alcanzar su máximo valor (o mínimo) en los estados objetivos del problema.
* Una búsqueda local elige los sucesores en pos de maximizar (o minimizar) una cierta función objetivo.

### Paisaje del espacio de estados

En un problema donde se debe maximizar la función objetivo, la búsqueda local explora el siguiente **paisaje**.



### Problemas de optimización

* Un **problema de optimización** tiene por objetivo maximizar una función objetivo f en un espacio de búsqueda X, es decir, max {f(x): x e X}
* Minimizar f es equivalente a maximizar -f.
* Los algoritmos de búsqueda local son muy útiles para resolver problemas de optimización.

### Estrategias de elección de sucesores

Con una formulación de estados completa, la búsqueda local siempre tiene estados sucesores para moverse… Entonces, ¿cuál elige?

## Búsqueda de ascensión de colinas

Hill-climbing search

La búsqueda de ascensión de colinas es la búsqueda local más simple.

Supongamos que queremos maximizar la función objetivo. En cada iteración se mueve al sucesor con **mayor** valor objetivo, siempre que mejore el valor objetivo actual, hasta atascarse.

En el paisaje del espacio de estados, se mueve hacia **arriba** hasta una **cima**.

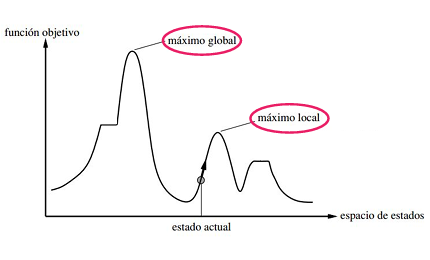
También es llamada búsqueda local **avara (greedy)**, ya que no mira más allá de los sucesores inmediatos.

### Máximos locales y globales

La ascensión de colinas se caracteriza por realizar un rápido ascenso por el paisaje del espacio de estados.

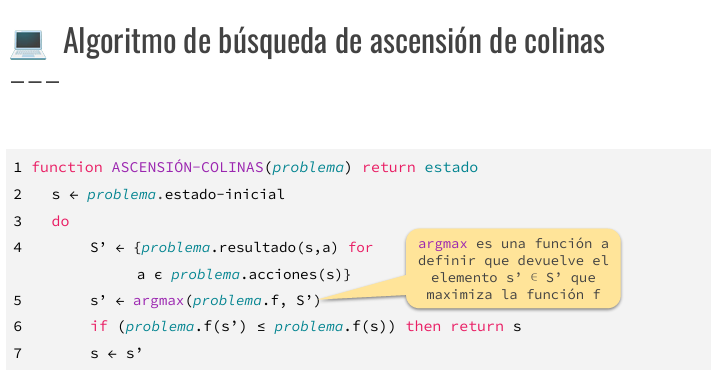
Pero una vez que llega a una cima, se **atasca**.

Dicha cima puede ser un **máximo global** o un **máximo local** de la función objetivo.



Dado un estado s, decimos que su valor objetivo f(s) es un **máximo local** si es mayor o igual al valor objetivo de todos sus sucesores, es decir, si: f(s’) <= f(s) para todo sucesor s’ de s.

Y decimos que es un **máximo global** si es un máximo entre todos los máximos locales.



### Características de ASCENSIÓN-COLINAS

* Muy bajo consumo de **memoria**.
* Se mueve siempre a un sucesor con valor objetivo **estrictamente mayor**.
* **Nunca cicla.**
* Se atasca en un **máximo** (local o global).
* En general, no hay garantía de que el estado retornado sea objetivo (**completitud**), ni que el máximo alcanzado sea global (**optimalidad**).

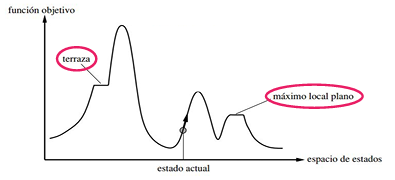
### Performance de ASCENSIÓN-COLINAS

La probabilidad y el tiempo requerido para alcanzar un máximo (local o global) depende del **problema** y de la **función objetivo**.

### Variantes de las Ascensión de colinas

Una **meseta** es un área del paisaje del espacio de estados donde la elevación es plana. Es decir, donde **el mejor estado sucesor tiene el mismo valor objetivo que el estado actual**.

Si no existe una salida ascendente, es un **máximo local plano**, de lo contrario es una **terraza.**



La ascensión de colinas también **se atasca al alcanzar una meseta**.

Tiene sentido retornar si estamos en un máximo local plano (es imposible seguir ascendiendo).

### Movimientos laterales

* Es posible moverse por una meseta mediante movimientos laterales, que consisten en moverse a un estado sucesor con el mismo valor objetivo que el estado actual.
* Permitir movimientos laterales en la ascensión de colinas puede permitir escapar de una meseta.

### Limitando movimientos laterales

* No siempre se necesita un solo movimiento lateral para escapar de una meseta.
* Podrían requerirse **múltiples** movimientos laterales consecutivos para escapar. Pero..
* **Permitir infinitos movimientos laterales consecutivos no es garantía de escape.**
  + Podríamos estar en un **máximo local plano** (y no en una terraza).
  + Podríamos estar **ciclando** entre estados con mismo valor objetivo.
  + En ambos casos, la ascensión de colinas **no terminaría jamás.**
* Para evitarlo, se suele **limitar** el número de movimiento laterales consecutivos

### Características de ASCENSIÓN-COLINA-LATERAL

* Muy bajo consumo de memoria.
* Se mueve siempre a un sucesor con valor objetivo **igual o mayor**.
* Se debe **limitar** el número de movimientos laterales consecutivos para garantizar **terminación**.
* Se atasca en un máximo (local, global o plano) y a veces también en una **terraza** (cuando no encuentra un escape con el número de movimientos laterales capaces).
* En general, no hay garantía de que el estado retornado sea objetivo (**completitud**), ni que el máximo alcanzado sea global (**optimalidad**)
* Típicamente se atasca con menor frecuencia en un máximo local (respecto a la ascensión de colinas clásica), pero suele necesitar más pasos para terminar.

### Otras variantes…

1. **Ascensión de colinas estocástica.**
   1. Elige el sucesor de forma **aleatoria**, entre todos aquellos que mejor el valor objetivo actual. También pueden asignarse probabilidades diferenciadas. Su convergencia suele ser más lenta, pero a veces encuentra mejores soluciones.
2. **Ascensión de colinas de primera opción.**
   1. Los estados sucesores se generan de a uno y se elige el primer sucesor encontrado cuyo valor objetivo es mejor que el actual. Es útil cuando los estados tienen muchos sucesores.
3. **Ascensión de colinas con reinicio aleatorio.**
   1. Realiza una serie de ascensiones de colinas. Cuando se atasca, se **reinicia** desde un estado inicial aleatorio. Retorna el estado con mejor valor objetivo encontrado entre todos los reinicios realizados.

Optimalidad. ✅ (con una cierta probabilidad) (el aleatorio)

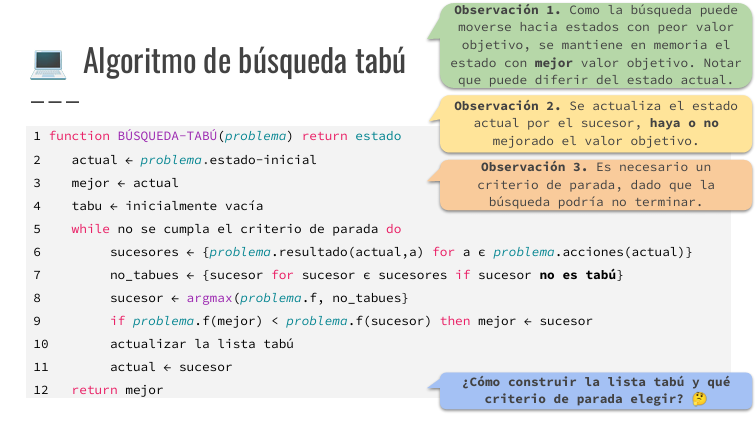
Si cada búsqueda de ascensión de colinas tiene probabilidad p de encontrar un máximo global, entonces se esperan 1/p reinicios para encontrar un máximo global con probabilidad cercana a 1.

## Búsqueda Tabú

Tabu Search

La búsqueda tabú es una búsqueda local que incorpora al algoritmo de ascensión de colinas ciertas mejoras para **escapar de máximos locales que no son globales**.

Se mueve **siempre** al sucesor con mejor valor objetivo, sea **mejor, peor o igual** que el actual.

Para evitar **ciclar**, mantiene una **memoria de corto plazo** con información de las últimas iteraciones que le permite evitar estados visitados recientemente. Esta estructura de datos se conoce como **lista tabú.**

### Lista tabú

¿Qué elementos se suelen almacenar en la lista tabú?

1. **Acciones**.
2. **Estados**
3. **Propiedades**

**Acciones** que podría revertir acciones realizadas recientemente

**Estados** que hayan sido visitados recientemente. En general, consume más tiempo y memoria. Los estados suelen ser más pesados que las acciones y compararlos consume más tiempo

**Propiedades** sobre atributos de los estados o de las acciones.

Los estados sucesores que verifiquen alguna de las propiedades de la lista tabú son ignorados.

Si los elementos de la lista tabú se almacenan para siempre podría tener un consumo de memoria **excesivo,** (dejaría de ser búsqueda local) mayor posibilidad de **atascarse**. Podría no existir un sucesor que no sea tabú.

La lista tabú debe funcionar como una **memoria de corto plazo.**

Existen dos enfoques:

1. **Limitar la capacidad de la lista tabú**. Si al agregar un nuevo elemento a la lista se excede su capacidad, entonces se borra de la lista el elemento más viejo. Es decir, la lista tabú funciona como una **cola**.
2. **Borrar un elemento de la lista tabú tras cierto límite de iteraciones**. Luego de agregar un nuevo elemento a la lista, se filtra la lista y se borran aquellos que hayan estado por más iteraciones del límite establecido. A este valor se lo conoce como **tenor de tabú**.

### Criterio de parada

1. Tras cierto número de **iteraciones totales** (o tiempo de CPU).
2. Tras cierto número de **iteraciones sin mejora** en el mejor valor objetivo.
3. Cuando la función objetivo sobrepasa cierto **valor umbral**.

### Componentes adicionales

En problemas difíciles se suelen incorporar a la búsqueda tabú algunos **componentes adicionales** para mejorar su efectividad

Entre los más comunes se encuentran:

1. Criterio de aspiración.
2. Diversificación.

### Criterio de aspiración

La lista tabú puede volverse demasiado restrictiva:

* Puede prohibir acciones atractivas, incluso cuando no hay riesgo de ciclar, o
* Puede conducir a un estancamiento de la búsqueda.

Un **criterio de aspiración** permite ignorar tabúes

El más sencillo consiste en permitir una acción, incluso si es tabú, siempre que el valor objetivo del estado sucesor sea mayor al del mejor estado encontrado hasta el momento (notar que no hay riesgo de ciclar)

### Diversificación

Los algoritmos de búsqueda local, en particular la búsqueda tabú, tienden a pasar la mayor parte del tiempo en una porción restringida del espacio de estados, ignorando quizás regiones más interesantes.

La diversificación es un mecanismo que fuerza la búsqueda en regiones inexploradas. Típicamente usa **memoria de largo plazo**.

Lo más sencillo es la **memoria de frecuencia,** que registra por cuántas iteraciones se preservaron algunos componentes de la solución.

A partir de esta info, existen dos estrategias de diversificación:

1. **De reinicio**. Reiniciar la búsqueda desde un estado con componentes poco frecuentes.
2. **Continua**. A la hora de elegir un estado sucesor, priorizar aquellos con componentes poco frecuentes (agregar un término de frecuencia a la función objetivo).

### Seteo de parámetros

* ¿Qué almacenar en la lista tabú: estados, acciones o propiedades?
* ¿Por cuánto tiempo almacenarlos: capacidad o tenor de tabú de la lista?
* ¿Cuándo parar: número de iteraciones, o número de iteraciones sin mejora, o tiempo límite, etc?
* …

Esto se determina experimentalmente. Ya que los tabu search son extremadamente sensibles al seteo de parámetros.

### Performance de BÚSQUEDA-TABÚ

* Bajo consumo de **memoria**, aunque mayor que ASCENSIÓN-COLINAS.
* No se **atasca** en máximos locales porque puede moverse a un sucesor con peor valor objetivo.
* Se mantiene una lista tabú para reducir el riesgo de **ciclar**.
* La **terminación** se garantiza con un **criterio de parada**.
* En general, no hay garantía de que el estado retornado sea objetivo (**completitud**), ni que el máximo alcanzado sea global (**optimalidad**)

## Búsqueda local en espacios continuos

En el mundo real, las magnitudes (longitud, masa, tiempo, etc) suelen asumir valores continuos.

En espacios de estados continuos, los estados tienen infinitos sucesores.

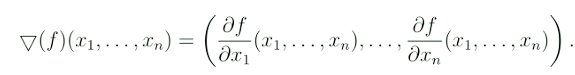
### Discretizar vs. no discretizar

Discretizar consiste en restringir el problema de forma tal que, para todo estado S, el conjunto ACCIONES(S) sea **finito**.

**Luego, podemos resolverlo con cualquiera de los algoritmos de búsqueda local descritos anteriormente.**

### No discretizar.

Gradiente

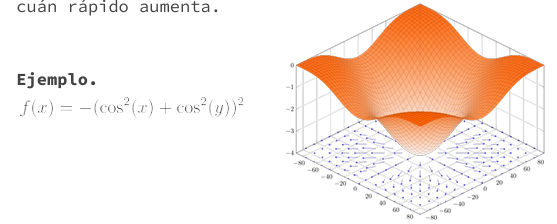
* Sea un campo escalar f : R^n -> R.
* El **gradiente** de f, notado ∇(f), es el campo vectorial 



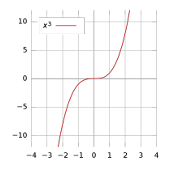
Donde es la **derivada parcial** de f respecto a xi

### Interpretación geométrica

Dado un punto (a1….an) e R^n, el vector ∇(f)(a1….an) muestra la dirección en la que hay que moverse estando en el punto (a1….an) para aumentar el valor de f lo más rápido posible y su módulo indica cuán rápido aumenta.



### Puntos estacionarios

* Dada una función diferenciable (derivable en cualquier dirección), un **punto estacionari**o es un punto del dominio donde todas las derivadas parciales son 0.
* Un punto estacionario puede ser:
  + Un máximo local,
  + un mínimo local,
  + o un punto de silla.
* Algunas veces, es posible calcularlos analíticamente.

### Aproximación numérica

* Desafortunadamente, no siempre es posible hallar los puntos estacionarios de forma analítica.
* Los máximos y mínimos locales se pueden aproximar numéricamente.

### Ascenso de gradiente (o ascenso por el gradiente)

El algoritmo de **ascensión de gradiente**:

1. Parte de un punto x del dominio.
2. En cada iteración se mueve al punto x <- x + α.∇(f)(x), donde α es una constante pequeña, llamada **tamaño de paso**.

Si se busca un mínimo local (descenso de gradiente) x <- x - α.∇(f)(x).



### Funciones no diferenciables

* Las funciones no diferenciables son más difíciles de tratar.
* No es posible obtener una fórmula cerrada para ∇(f).
* Si el gradiente se puede calcular localmente, entonces todavía es posible aplicar el ascenso de gradiente.

# Búsqueda entre adversarios

### Búsqueda entre adversarios

Los problemas de búsqueda entre adversarios, también llamados juegos, involucran entornos competitivos. Nos limitamos a:

* Dos jugadores.
* Por turnos.
* Suma cero.
* Información perfecta

### Juegos

Formalmente, los juegos son un tipo de problema de búsqueda con los siguientes 6 elementos.

1. **S0** . **Estado inicial**.
2. **JUGADOR(S)**. Determina qué jugador tiene el turno en ese estado. Los jugadores se llaman MAX y MIN, siendo MAX quién juega primero.
3. **ACCIONES(S)**. Devuelve el conjunto de **acciones** legales en ese estados
4. **RESULTADO(S,A)**. El **modelo de transiciones**, que define el estado resultado de aplicar esa acción en ese estado.
5. **TEST-TERMINAL(S)**. Determina si el juego está terminado en ese estado. Donde sí está finalizado se llama **estado terminal**.
6. **UTILIDAD(S,P)**. Una **función de utilidad** que asigna un valor numérico (recompensa) a cada jugador P en un estado terminal S.

Un juego es de suma cero si la suma de las utilidades de los jugadores es la misma para todo estado terminal.

### Árbol de juego

El estado inicial, las acciones y el modelo de transiciones definen un **árbol de juego**, donde los nodos representan estados y las ramas representan acciones, similar a un árbol de búsqueda.

### Estrategia óptima

En un juego, el objetivo es hallar una **estrategia** que especifique:

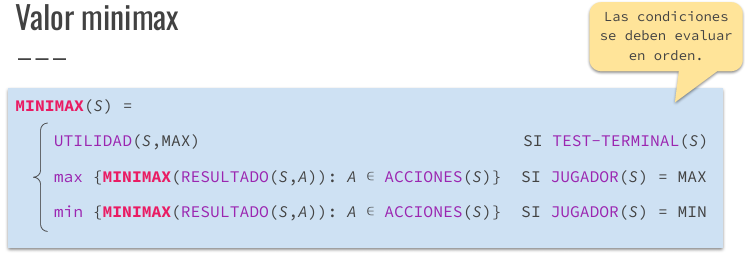
1. La acción de MAX en el estado inicial.
2. Luego, la acción de MAX en los estados que resulten de cada posible respuesta de MIN.
3. Luego, la acción de MAX en los estados que resulten de cada posible respuesta de MIN a las acciones anteriores, y así sucesivamente.

Una **estrategia óptima** produce un resultado al menos tan bueno como cualquiera otra estrategia cuando se juega contra un oponente **infalible**.

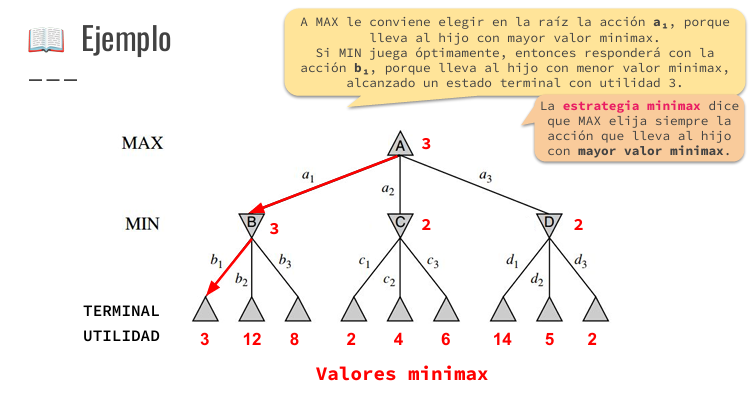
### Valor minimax

Dado un árbol de juego,

* El **valor minimax** de un nodo es la “utilidad” (para MAX) de estar en el estado correspondiente, **asumiendo que ambos jugadores juegan óptimamente** desde allí al final del juego
* **La estrategia óptima puede determinarse a partir del valor minimax de cada nodo**.



* El valor minimax de un nodo con un estado terminal es su utilidad.
* MAX prefiere moverse a un nodo con mayor valor minimax.
* MIN prefiere moverse a un nodo con menor valor minimax.



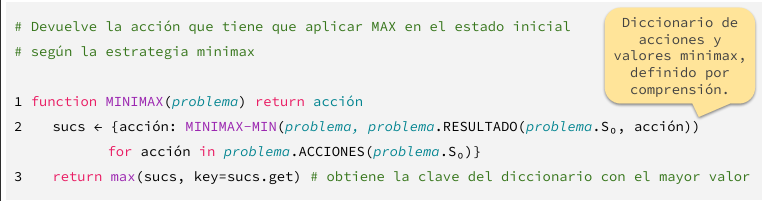
### Estrategia minimax

* La estrategia minimax es óptima, asumiendo que MAX y MIN juegan óptimamente.
* ¿Y si MIN no juega óptimamente? Entonces a MAX le irá aún mejor.
* Pero otras estrategias contra oponentes subóptimos pueden ser incluso mejores que minimax, aunque necesariamente son peores contra oponentes óptimos.

### Implementación del algoritmo minimax

* Dado un nodo que no es hoja, para determinar su valor minimax es necesario conocer el de sus hijos.
* Es decir, hay que llamarse **recursivamente** en cada uno de sus hijos y elegir el máximo o mínimo valor, según si es el turno de MAX o MIN respectivamente.





### Performance de MINIMAX

* Hace una exploración primero en profundidad (recursiva) del árbol de juego completo.
* Sean m la altura del árbol y b el factor de ramificación.
* **Consumo de tiempo**. Genera b^m nodos.
* **Consumo de memoria**. Se necesitan almacenar b\*m llamadas recursivas.
* **Para juegos reales, este consumo de tiempo es completamente impráctico**. Pero el algoritmo minimax sirve como base para algoritmos más sofisticados y para el análisis teórico.

## Poda alfa-beta

En determinadas circunstancias, es posible computar la estrategia minimax **sin mirar todos** los nodos del árbol de juego.

Basta con **podar** aquellos nodos para los cuales se tenga evidencia suficiente de que **no influyen** en la decisión final.

Esta técnica se llama **poda alfa-beta**

Si MAX tiene una opción n, pero más arriba en el árbol tiene una opción m mejor que n, entonces **nunca será alcanzado en una jugada real** (siempre asumiendo que MAX y MIN juegan óptimamente).

Si se tiene suficiente información sobre m y n para llegar a esta conclusión, entonces es posible podar a n.

Supongamos que:

1. Computamos el valor minimax de m y es un número alfa. Es decir, MAX tiene una opción de valor alfa.
2. Más tarde, mientras computamos el valor minimax de n, logramos concluir que es menor a alfa.

Entonces, MAX nunca elegirá moverse a n y podemos podarlo **de inmediato**, sin necesidad de computar su valor minimax de forma exacta.

Durante la ejecución, se mantienen dos parámetros (alfa, beta) con las cotas de los valores minimax encontrados previamente.

alfa: valor de la mejor opción para MAX encontrada hasta ahora en el camino actual.

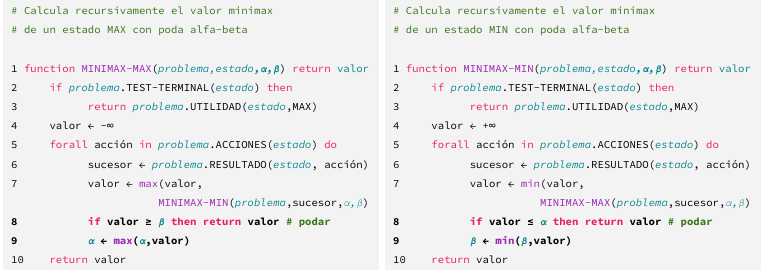
beta: valor de la mejor opción para MIN encontrada hasta ahora en el camino actual.

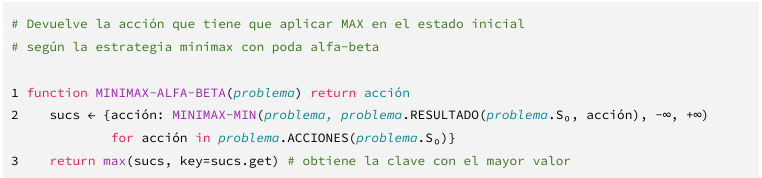
En general,

* Se puede aplicar a árboles de cualquier altura.
* Puede podar subárboles enteros y no solo hojas.

### Implementación de la poda alfa beta

* Las funciones MINIMAX-MAX y MINIMAX-MIN se aumentan con dos nuevos parámetros: α y β.
* MINIMAX-MAX:
  + Actualiza α si el valor minimax del nodo es más grande que α.
  + Poda si uno de sus hijos tiene valor minimax más grande que **β.**
* MINIMAX-MIN:
  + Actualiza β si el valor minimax del nodo es más chico que β.
  + Poda si uno de sus hijos tiene valor minimax más chico que α.





### Performance de MINIMAX-ALFA-BETA

* Devuelve lo mismo que MINIMAX sin poda. Es decir, la estrategia óptima.
* Menor consumo de tiempo que MINIMAX sin poda, aunque sigue siendo exponencial en la altura.
* Es muy dependiente del orden en el que se examinan los sucesores.

## Decisiones imperfectas en tiempo real

La profundidad en el árbol de juego a la que necesita llegar el algoritmo minimax a menudo no es práctica.

Típicamente las acciones deben ser realizadas en tiempos razonables, unos minutos como mucho.

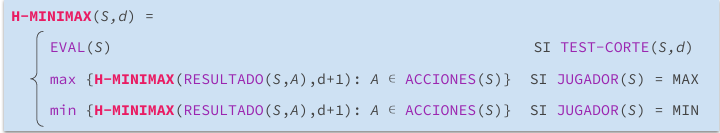
¿Cómo tomar buenas decisiones en pocos minutos?

Una alternativa es cortar la búsqueda **anticipadamente** en un cierto nivel de profundidad y devolver la mejor decisión para ese árbol de juego parcial.

### Funciones evaluación

* Los estados no-terminales en un cierto nivel de profundidad se interpretan como terminales.
* A dichos estados, se le aplica una **función de evaluación** que retorna una estimación de la **utilidad esperada**.
* El momento en el que se corta la búsqueda se determina con una función de **test de corte**.

Así, tenemos la siguiente heurística minimax para un estado S en una profundidad d:



* ¿Cómo diseñar buenas funciones de evaluación, EVAL?
* Depende del juego (al igual que sucede con las heurísticas en las búsquedas informadas).
* Veremos dos técnicas generales para diseñarlas.

### Diseño de funciones evaluación - Primera técnica

Se agrupan en una misma categoría (o clase de equivalencia) aquellos estados que tengan los mismos valores en ciertas características. (ejemplo ajedrez quedan 2 peones vs. 1 peón).

Luego, para cada categoría, se computa un **valor esperado** que refleje la proporción de sus estados que conducen a triunfos, empates y pérdidas.

**En la práctica, este análisis suele requerir muchas categorías y demasiada experiencia para estimar correctamente las probabilidades.**

### Diseño de funciones evaluación - Segunda técnica.

Se eligen ciertas **características** de los estados y se les asigna una contribución numérica, que luego se **combinan** para encontrar el valor total. (Ejemplo a cada pieza de ajedrez se le asigna un valor material, que después sumado te da el valor total.)

Las contribuciones se pueden sumar directamente para obtener la evaluación de un estado. Así, la función de evaluación es una función ponderada lineal (weighted linear function):

Donde cada wi es un peso y cada fi es una característica.

Esta fórmula asume que la contribución de cada característica es **independiente** de las demás. Funciones más poderosas pueden incorporar términos no lineales.

### Performance

* Ya no garantiza encontrar la estrategia óptima.
* Se puede derrotar a oponentes subóptimos con una buena función de evaluación.
* El consumo de tiempo se puede reducir drásticamente con un test de corte apropiado.
* Mientras más capas del árbol de juego se miren, más acertadas serán las decisiones pero mayor el consumo de tiempo.

# Problemas de satisfacción de restricciones

Problema es del coloreo del mapa de australia.

Objetivo. Colorear cada región de Australia con rojo, verde o azul de manera que ningún par de regiones limítrofes tenga un mismo color.

Formularlo de forma incremental, lleva al inconveniente de que podemos pintar una región del mismo color que una región limítrofe. Tales estados nunca conducirán a estados objetivos ya que no es posible cambiar el color de una región ya pintada.

Se reformula y se le da restricciones a las acciones con una función COL(s1,...,sj) que devuelve el conjunto de colores con los que pintaron las regiones vecinas a j+1.

COL(s1,...,sj) = {si: i e {1,..,j}, las regiones i y j+1 son vecinas}

Esto es demasiado específico y lleva a complejizarse mucho.

## Problema de Satisfacción de Restricciones

Constraint Satisfaction Problem (CSP)

Un problema de satisfacción de restricciones (CSP) se compone de:

1. Un conjunto de variables. X = {X1,..,Xn}
2. Un conjunto de dominios. D = {D1,..,Dn}. Cada dominio Di = {vi1,..vik} es el conjunto de valores permitidos para la variable Xi.
3. Un conjunto de restricciones que especifica las combinaciones de valores permitidas. C = {C1,..,Cr}

### Restricciones

Cada restricción Ci de C es un par <Yi, Ri> donde:

* Yi es una tupla de variables participantes en la restricción,
* Ri es una relación con los valores que esas variables pueden tomar.

Por ejemplo si las variables X1 y X2 tienen dominio D1 = D2 = {0,1}, entonces una restricción que diga que X1 y X2 deben tener valores diferentes se escribe:

<(X1,X2) , X1 != X2> o por extensión.

Si necesitamos establecer precedencia entre las variables, es decir, primero se tiene que hacer una acción y después se puede hacer otra. Se llama restricción de precedencia.

En el ejemplo del tiempo para instalar los trenes, ruedas, tuercas y tazas, si necesito poner una restricción que primero se hacer un tren y después el otro, es una restricción disyuntiva, ya que tengo hacer primero el delantero y después el trasero o viceversa.

### Soluciones

* Una asignación asocia un valor o algunas o todas las variables. {Xi = vi, Xj = vj, ..} Cada variable debe tomar un valor de su dominio.
* Una asignación es completa si asigna un valor a todas las variables, de lo contrario es parcial.
* La asignación es consistente (o legal) si no viola ninguna restricción, de lo contrario es inconsistente (o ilegal).
* Una solución es una asignación completa y consistente.

### Clasificación de variables

Según su dominio, una variable se clasifica en:

* **Discreta finita**. Di = {v1,..,vn}, para algún natural n.
* **Discreta infinita**. Por ejemplo, Di = Naturales.
* **Continua**. Por ejemplo, Di = Reales.

En variables con dominios infinitos no es posible en general expresar las restricciones por extensión.

Nos limitaremos a variables discretas finitas.

### Clasificación de restricciones

* **Unaria**. Restringe el valor de una única variable. Por ejemplo: <(S), S != Verde>
* **Binaria**. Relaciona dos variables. Por ejemplo: <(S,N), S !=N>
* **Global**. Relaciona un número arbitrario de variables. Por ejemplo: <(W,NS) , Alldiff(W,N,S)> donde Alldiff(W,N,S) significa que las variables W,N,S deben tener valores mutuamente diferentes.

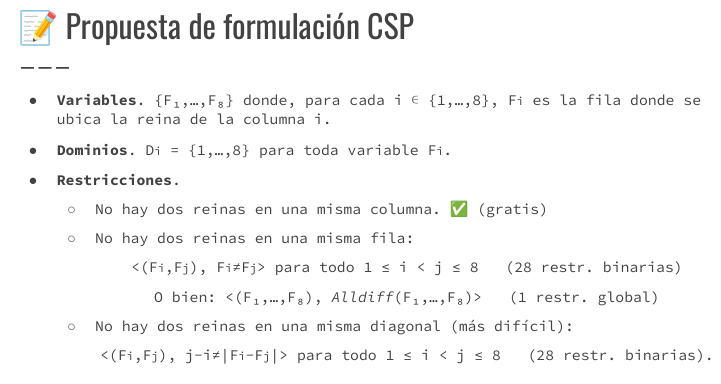
Toda restricción unaria puede eliminarse de un CSP modificando adecuadamente el dominio de la variable involucrada.

En general, es posible transformar una restricción global en un conjunto de restricciones binarias.

Todo CSP con restricciones unarias, binarias o globales se puede transformar en un CSP equivalente que tiene únicamente restricciones binarias.

### Grafo de restricciones

Los CSP suelen visualizarse con **grafos de restricciones** - donde los nodos son variables y los arcos conectan todo par de variables que participan en una restricción binaria.

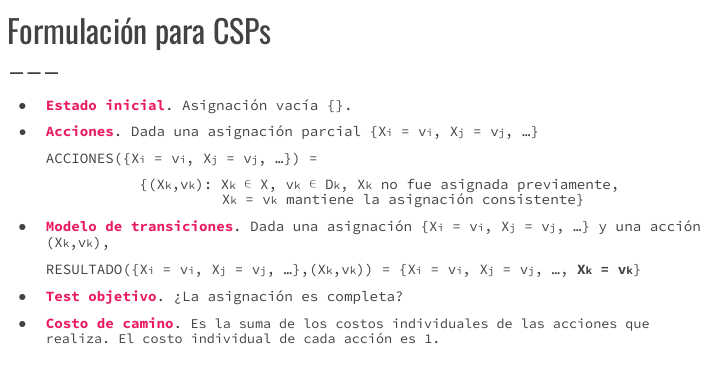


## Búsqueda clásica

### Formulación para CSPs

Vamos a convertir un CSP general en un problema de búsqueda.

* Los **estados** son todas las asignaciones consistentes (parciales o completas).
* Las acciones son pares (Xi, vi) e X x Di que representa asignar a la variable Xi el valor vi de su dominio.

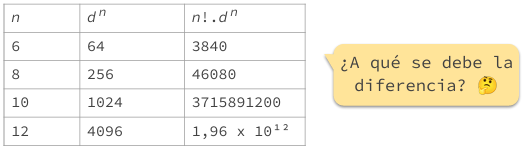


### Tamaño del árbol de búsqueda

De manera más general, considerar un CSP con n variables, cada una con un dominio de tamaño d.

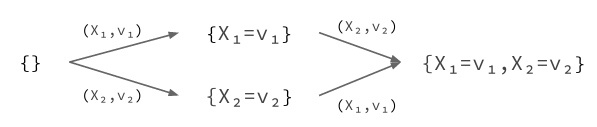
* El número de asignaciones completas está dado por el producto del cardinal del dominio de todas las variables: |D1| \* |D2| \* … \* |Dn| = d\*d\*...\*d = d^n
* En el árbol de búsqueda el factor de ramificación en la raíz es n\*d, en el siguiente nivel es (n-1)\*d y así… Este árbol tendría n!\*d^n hojas.

Supongamos que d = 2 ( número de valores) y comparemos estos números al incrementar n (número de variables):



y la diferencia es más significativa con el aumento de d.

La diferencia se debe a la gran cantidad de caminos redundantes.



### Conmutatividad

* Los CSPs tienen la propiedad de **conmutatividad**: si se aplican las mismas acciones pero en otro orden, se obtiene la misma asignación.
* Entonces, basta con definir un **orden** para las variables.
  + En el nivel 0 (la raíz) se asigna un valor a X1.
  + En el nivel 1 se asigna un valor X2.
  + Y así sucesivamente.. hasta el nivel n-1 donde se asigna un valor a la última variable Xn.

## Búsqueda vuelta atrás

Backtracking search

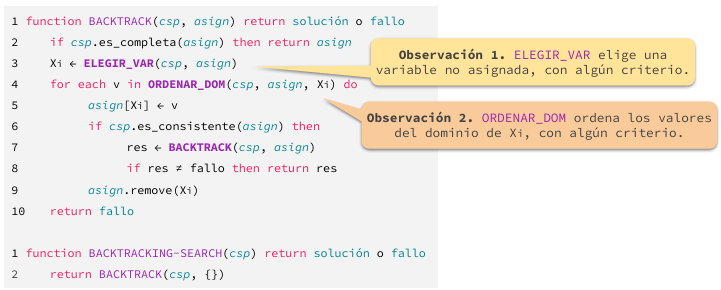
La búsqueda vuelta atrás es básicamente un **DFS** con un **orden** predefinido para las variables del CSP.

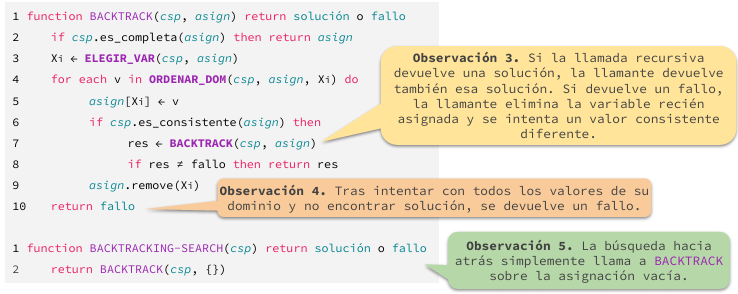
En cada nivel del árbol de búsqueda se asigna un **valor consistente** a la siguiente variable de acuerdo al orden.

**Retrocede** cuando a una variable no le quedan valores consistentes para asignar.

### Algoritmo de búsqueda vuelta atrás

* Veremos una implementación recursiva (no almacena explícitamente una frontera).
* Se define la función BACKTRACK que toma una asignación parcial, elige un valor consistente para una variable no asignada y se llama recursivamente.
* Esto se repite hasta encontrar una asignación completa.
* Si a una variable no le quedan valores consistentes para asignar, el algoritmo devuelve un fallo y la llamada recursiva termina.
* Tras el fallo, la búsqueda retrocede a la última llamada y se elige un valor, consistente diferente para asignar a la variable.





* Para todo CSP, BACKTRACKING-SEARCH usa implícitamente **el mismo estado inicial, función de acciones, modelo de transiciones y test objetivo**.
* **No necesita funciones específicas del problema a resolver.**
* Mismo **consumo de tiempo y memoria que DFS**.
* Garantiza **completitud** si los dominios son finitos.

## Búsqueda local

Otra alternativa para resolver un CSP general es mediante búsqueda local. Necesitamos:

1. Formulación de estados completa.
2. Función objetivo que alcance su máximo valor en las soluciones.

### 1. Formulación de estados completa

Los **estados** son las **asignaciones completas** (consistentes o inconsistentes), es decir, todas las variables ya tienen asignado un valor de su dominio.

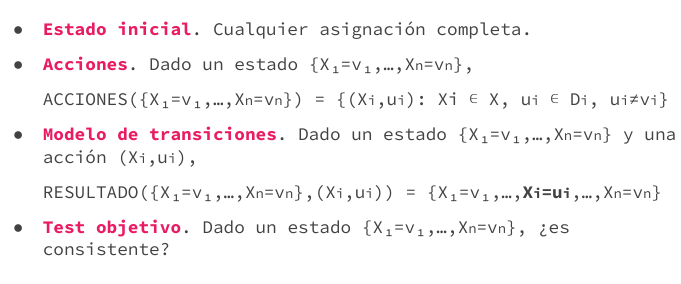
El número total de estados es: Π |Di|

es decir, el producto de los cardinales de los dominios (siempre que sean finitos).

Las **acciones** son los pares (Xi, vi) e X . Di

que representan **reasignar** a la variable Xi con el valor vi de su dominio.

El número de sucesores en cada estado es: Σ |Di| - 1.

Es decir, es posible **reasignar** a cualquier variable con cualquier valor de su dominio diferente de su valor actual..

### 2. Función objetivo

* El objetivo es encontrar una asignación consistente.
* Es decir, una asignación donde el número de restricciones violadas sea 0.
* Definimos la función f tal que, para toda asignación s, f(s) es el número de restricciones violadas.
* El objetivo es minimizar f o equivalentemente, maximizar -f.
* Notar que 0 es el mínimo global y f(s) = 0 si y sólo si s es una solución.

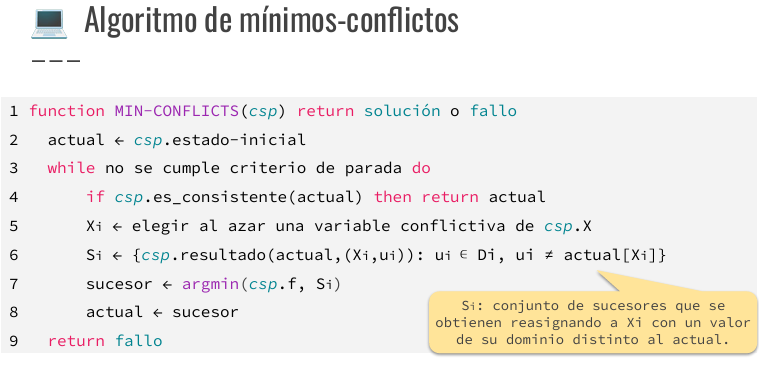
## Heurística de mínimos-conflictos

Min-conflicts heuristic

La heurística de mínimos-conflictos es un algoritmo de búsqueda local para resolver CSPs.

Usa la formulación de estados completa y la función objetivo presentadas anteriormente.

En cada iteración, se elige al azar una variable conflictiva (participante en alguna restricción violada) y se le asigna el valor que minimiza el número de restricciones violadas.



### Performance de MIN-CONFLICTS

* Es sorprendentemente eficaz para muchos CSPs.
  + Resuelve el problema del millón de reinas en un promedio de 50 pasos (luego de la asignación inicial).
* El paisaje de espacio de estados de un CSP tiene usualmente muchas mesetas.
* Las técnicas vistas anteriormente para búsqueda local pueden ayudar a escapar de estas mesetas: movimientos laterales y lista tabú.

## Backtracking mejorado

* La búsqueda hacia atrás elige una variable no asignada y recorre los valores de su dominio, con algún criterio.



* El criterio para recorrer el dominio podría acelerar el hallazgo de soluciones.
* Los criterios más sencillos son:
  + Orden predefinido: alfabético, natural (1,2,3) ,ect.
  + Orden aleatorio
* Ninguno de ellos aprovecha propiedades de los CSP.

### 1. Elección de variable

**Criterio de mínimos valores restantes (MVR)**. La próxima variable a asignar es la que tiene menor número de valores consistentes en su dominio.

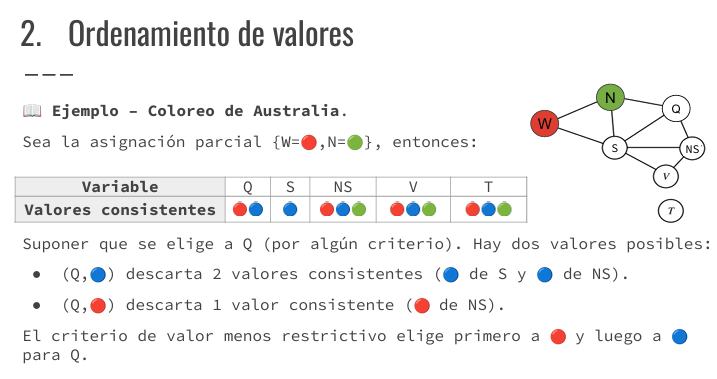
**Criterio de grado**. La próxima variable a asignar es la que está relacionada con el mayor número de variables no asignadas.

Suele usarse como **regla de desempate** en la heurística de mínimos valores restantes.

### 2. Ordenamiento de valores

Elegida la variable, ahora hay que elegir un valor de su dominio.

**Criterio de valor menos restrictivo**. Dada una variable, el próximo valor a elegir es el que inhabilita el menor número de valores en las demás variables no asignadas.



* Con estos criterios, ya podemos implementar un algoritmo de búsqueda hacia atrás bastante digno. Ejemplo el de las 1000 reinas.
* **No precisan conocimiento específico del problema** (a diferencia de heurísticas en las búsquedas informadas).
* Son **generales** (aplican a cualquier CSP).

### 3. Comprobación hacia adelante

Luego de asignar una variable Xi, en las variables no asignadas relacionadas con Xi pueden quedar valores que hagan a la asignación inconsistente.

* Comprobación hacia adelante. Siempre que se asigne una variable Xi, se mira cada variable no asignada Xj que esté relacionada con Xi por una restricción y se suprime del dominio de Xj cualquier valor que sea inconsistente con el valor elegido para Xi.
* Si una variable no asignada se queda con **dominio vacío**, entonces la asignación es inconsistente y la búsqueda debe **retroceder**.

Los dominios se van achicando a medida que asignamos variables

* Propaga información (de variables asignadas a no asignadas).
* Reduce el tamaño de los dominios y por ende el consumo de tiempo.
* Detecta inconsistencias (algunas).
* Simplifica el cómputo de la **heurística de MVR**.

### 4. Propagación de restricciones

La comprobación hacia adelante detecta tempranamente muchas inconsistencias, pero no todas.

La comprobación hacia adelante no mira lo suficientemente adelante.

* El problema es que la comprobación hacia adelante no mira lo suficiente adelante.
* Es necesario propagar las restricciones entre variables no asignadas.

### Arco consistencia

Sean dos variables Xi y Xj relacionadas por una restricción binaria <(Xi, Xj), R>.

Decimos que Xi es arco-consistente con Xj si:

para todo Vi e Di, existe Vj e Dj tal que (Vi,Vj) e R

Decimos que Xj es arco-consistente con Xi si:

para todo Vj e Dj, existe Vi e Di tal que (Vi,Vj) e R

Es decir, para todo valor del dominio de la primera variable, existe un valor en el dominio de la segunda variable que es consistente con el valor de la primera (no viola la relación R).

Es como un tipo de inyectividad. Si el dominio de la variable de la izq tiene para cada valor un valor en el dominio de la derecha, se dice que es arco-consistente de izq a der, izq es arco-consistente con der, y si sucede al revés, se dice lo contrario.

* Si una variable Xi no es arco consistente con otra Xj, entonces es posible restringir el dominio de Xi para forzar la arco consistencia.
* La idea es eliminar los valores que violen la condición de arco consistencia.

### 4. Propagación de restricciones

* La **comprobación hacia adelante** y la **arco consistencia** son formas de **propagación de restricciones**.
* La comprobación hacia adelante fuerza que las variables relacionadas sean arco consistentes con la variable asignada.
* Pero no fuerza que esas variables sean arco consistentes con otras variables no asignadas.
* **Mantenimiento de la consistencia de arco (MCA)**. Siempre que se asigne una variable, se impone repetitivamente la arco consistencia entre todas las variables.
* El algoritmo usado para imponer arco consistencia se denomina AC-3 y utiliza una cola.
* MCA detecta anticipadamente **más inconsistencias** que la comprobación hacia adelante, pero es **considerablemente más costoso**.
* En algunos problemas, MCA puede mejorar el **consumo de tiempo**, en otros puede empeorar debido a la **sobrecarga computacional**.

### 5. Restricciones globales

Las restricciones globales permiten detectar inconsistencias de manera más eficiente que los métodos generales presentados hasta ahora.

* La restricción <(X1,...,Xm), Alldiff(X1,...,Xm)> representa que las variables X1,..,Xm deben tener valores distintos.
* La restricción <(X1,..,Xm), Atmost(K, X1,..,Xm)> representa que los valores de X1,..,Xm suman a lo sumo K, siendo K alguna constante numérica. Es decir, <(X1,..,Xm), X1 + … + Xm <= K)>

### Restricciones globales - Alldiff

<(X1,...,Xm), Alldiff(X1,...,Xm)>

**Test de inconsistencia**. Si en los dominios hay n posibles valores distintos para X1,..,Xm y n < m, entonces la restricción no se puede verificar.

### Restricciones globales - Atmost

<(X1,..,Xm), Atmost(K, X1,..,Xm)>

**Test de inconsistencia**. Si la suma del menor valor del dominio de cada una de las variables X1,..,Xm es mayor a K, entonces la restricción no se puede verificar.

## Estructura de los problemas

La dificultad para resolver un CSP está fuertemente relacionada con la **estructura** del grafo de restricciones.

Una de las estructuras más importantes son las **componentes conexas** del grafo.

### Ventajas

Suponer que un CSP tiene n variables con d posibles valores y que se divide en k subproblemas independientes, cada uno de ellos con n/k variables.

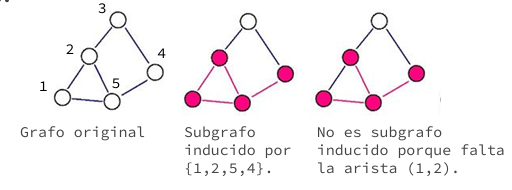
La cantidad de asignaciones completas en el problema original es: d^n

La cantidad de asignaciones completas en los subproblemas es: k.d^(n/k)

Para n=80, d=2 y k=4, pasamos de 1,2x10^24 a 4.194.304 asignaciones completas.

### Subgrafo inducido

Dado un grafo G=(V,E) y un subconjunto U ⊆ V, se define el subgrafo inducido G[U] como aquél que tiene a U como conjunto de vértices y toda arista de G con extremos en U es una arista de G[U].



### Componente conexa

* Una **componente conexa** de un grafo es un subgrafo inducido que es:
  + **Conexo**. Todo par de vértices están conectados por un camino.
  + **Maximal**. Se desconecta al agregar cualquier otro vértice del grafo.
* Cada componente conexa del grafo se corresponde con un subproblema independiente.
* Existen algoritmos para encontrar las componentes conexas de un grafo, que están basados en DFS: algoritmo de Kosaraju, algoritmo de Tarjan, entre otros.

# Búsqueda inspiradas en la naturaleza

## Recocido Simulado

Simulated Annealing (SA)

El recocido es un tratamiento térmico que permite alterar la estructura molecular de un material para mejorar sus propiedades mecánicas.

1. **Calentar** un material por encima de su temperatura de recristalización.
2. **Mantener** la temperatura por un período de tiempo.
3. **Enfriar** progresivamente hasta alcanzar la temperatura ambiente.

El recocido simulado es un algoritmo de búsqueda local con un mecanismo para escapar de máximos locales inspirado en la etapa de enfriamiento del proceso de recocido.

* Permite moverse a un vecino que empeore la función objetivo con una cierta **probabilidad**.
* Esta probabilidad decrece a medida que el sistema se **enfría**.
* Es decir, los “malos” movimientos son más probables al comienzo y se vuelve menos probables a medida que la temperatura baja.

La idea es:

1. Comienza en un estado inicial con una temperatura inicial.
2. En cada iteración:
   1. La temperatura **disminuye**.
   2. Se elige al **azar** un estado sucesor s’ del estado actual s.
   3. Se acepta s’ como un nuevo estado actual con un cierta **probabilidad**
3. El algoritmo se detiene cuando la temperatura alcanza un valor final, típicamente 0.

### Esquema de recocido

El **esquema de recocido** es una función que determina la temperatura en función del tiempo (número de iteraciones).

Típicamente, se da en función de una **temperatura inicial** t0 y de un **factor de enfriamiento** c e (0,1).

Es decir, la temperatura inicialmente es t0 y en cada iteración se enfria ti = c\*ti-1 , en la iteración i, ti = t0 \* c^i este esquema de recocido es de tipo **exponencial**.

### Probabilidad de aceptación

La función de **probabilidad de aceptación** P(s, s’, t) determina la probabilidad de aceptar el estado sucesor s’ de s como nuevo estado actual cuando la temperatura es t.

Debe cumplir las siguientes dos propiedades:

1. Si f(s) < f(s’), es decir, s’ es mejor que s, entonces: P(s, s’, t) = 1. Esto quiere decir que siempre que se acepta a s’, independientemente de la temperatura.
2. Si f(s) >= f(s’), es decir, s’ no es mejor que s, entonces: 0 < P(s, s’,t) <= 1, y tiene a 0 cuando t disminuye. Esto quiere decir que se acepta a s’ con una cierta probabilidad. Pero que será menos probable aceptarlo a medida que se enfría el sistema.

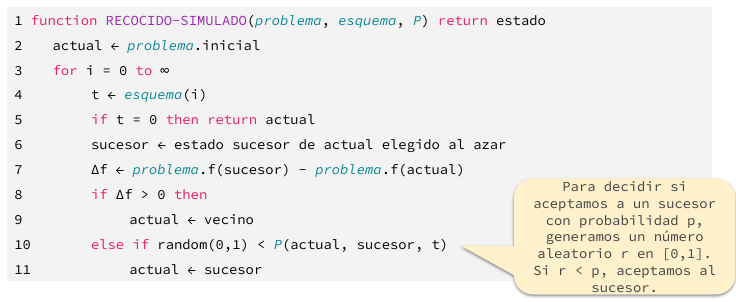
La función de probabilidad de aceptación para sucesores que empeoren el estado actual mayormente es:

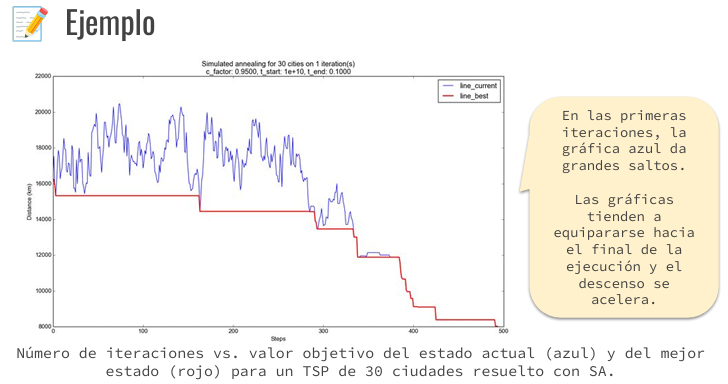
Esto porque e^x es una función que cuando x <= 0 está acotada en (0, 1]. Además como t disminuye, el gráfico tiende a 0.

La temperatura inicial t0 se suele determinar de manera que la probabilidad de aceptar un estado que sea peor en un w% al estado inicial s0 sea igual a 0.5.

El valor más adecuado para w y c (factor de enfriamiento) se determina por **experimentación.**







## Algoritmo Genético

Genetic Algorithm (GA)

Los **algoritmos genéticos** se inspiran en la teoría evolutiva de la **selección natural**.

* Se inicia con una **población** de individuos.
* En cada iteración, se **seleccionan** los mejores individuos de la generación actual.
* Los genomas de los individuos seleccionados se **cruzan** y se **mutan** para formar la siguiente generación.
* Típicamente, el algoritmo termina cuando se alcanza cierto número de generaciones.

### Población inicial

Un GA comienza con una **población inicial** formada por un conjunto de k estados (o **individuos**) generados aleatoriamente.

El genoma de cada individuo se representa con una string sobre un alfabeto finito.

### Función de fitness

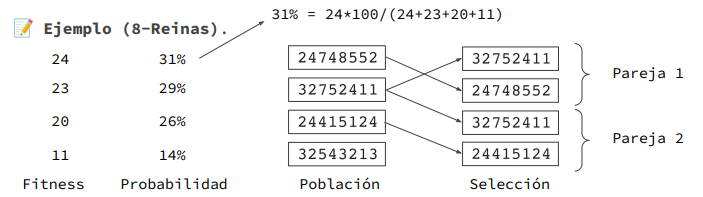
Cada individuo se califica con una **función de fitness** ( o función objetivo).

La función de fitness debe devolver valores más altos para los mejores individuos.

### Selección

La **selección** consiste en elegir al azar pares de individuos de la generación actual para reproducirlos.

Usualmente, la probabilidad de que un individuo sea elegido es directamente proporcional a su valor de fitness.



Para elegir a un individuo al azar con probabilidades diferenciadas generamos un número aleatorio r en [0,1]. Luego si 0 <= r <= 0.31, individuo 1

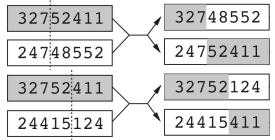
0.31 < r <= 0.60, individuo 2

0.60 < r <= 0.86, individuo 3

0.86 < r <= 1, individuo 4.

### Cruzamiento

El **cruzamiento** consiste en generar nuevos individuos a partir de la información genética de sus padres.

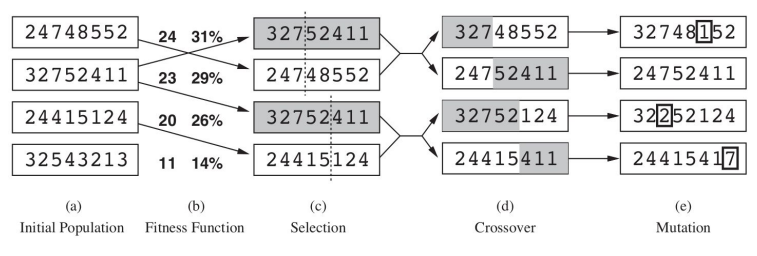
**Cruzamiento clásico**. Se elige una posición i aleatoria de la string y se combinan los primeros i caracteres del primer individuo con los caracteres restantes del segundo individuo.

### Mutación

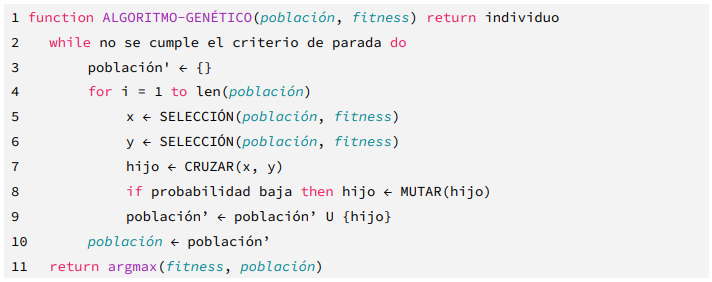
La **mutación** consiste en realizar con una probabilidad baja un cambio aleatorio en la información genética del nuevo individuo.

**Mutación clásica**. Se elige una posición i aleatoria de la string y se cambia su valor por un valor aleatorio.





Algoritmo genético



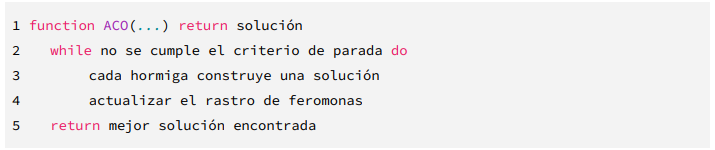
**Inicialmente, la población es bastante diversa**, luego el cruzamiento produce individuos muy diferentes a sus padres. **Hacia las últimas generaciones, los individuos son muy similares**, luego el cruzamiento produce individuos muy parecidos a sus padres.

## Optimización por colonia de Hormigas

Ant Colony Optimization (ACO)

La optimización por colonia de hormigas se inspira en el rastro de **feromonas** que usan las hormigas como medio de comunicación.

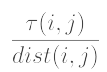
1. En cada iteración, una población de hormigas artificiales construye soluciones al problema guiadas por un rastro de feromonas.
2. Antes de pasar a la siguiente iteración, se ajusta la cantidad de feromonas por **depósito** y la **evaporación**, para reflejar la experiencia acumulada durante la última iteración.



### Construcción de soluciones

Una población de m hormigas construye soluciones de forma **incremental** y **estocástica**.

* Cada hormiga empieza con una solución parcial inicial.
* En cada paso de construcción, la hormiga elige un componente de solución al azar y aumenta su solución parcial.
* La probabilidad de elegir un componente de solución depende del rastro de feromonas y de la función de evaluación.

Cada hormiga empieza en una ciudad aleatoria y, en cada paso de construcción, la hormiga viaja a una ciudad no visitada. Estando en i, la probabilidad de moverse a j es directamente proporcional a: 

donde t(i,j) es la cantidad de feromona y dist(i,j) es la distancia de la arista (i,j).

Es decir, **la probabilidad es más grande cuando hay más feromonas y menor es la distancia**.

### Depósito y evaporación

La cantidad de feromonas se actualiza a fin de que los mejores componentes de solución sean más deseables para las hormigas en las futuras iteraciones.

* **Depósito**. Agrega feromonas a los componentes de solución elegidos por las hormigas. Los mejores componentes reciben más feromonas.
* **Evaporación**. Decrece a lo largo del tiempo las feromonas depositadas por hormigas anteriores.

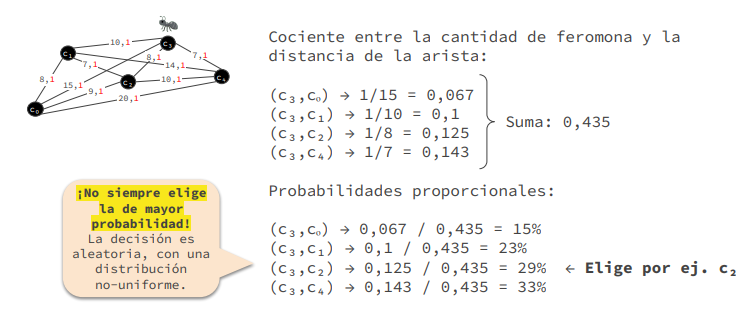
La cantidad de feromonas t(i,j) de cada arista (i,j) se actualiza:

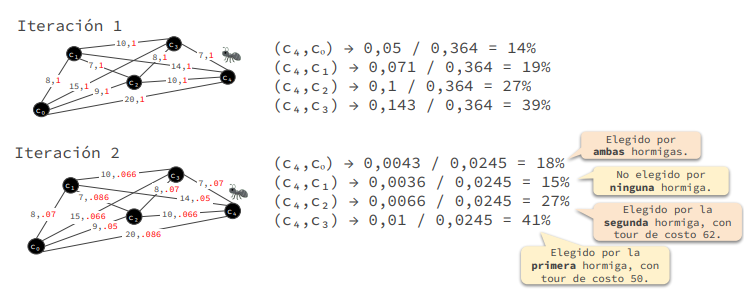


donde p e (0,1) y dist(tourk) es el largo del tour encontrado por la k-ésima hormiga.

Es decir, se evapora el p% de la feromona acumulada y, por cada aparición de la arista en el tour de una hormiga, se deposita una cantidad de feromona inversamente proporcional al costo del tour.

Así, **aristas usadas por muchas hormigas y contenidas en tours cortos reciben más feromonas y por ende son más probables a ser elegidas en futuras iteraciones**.





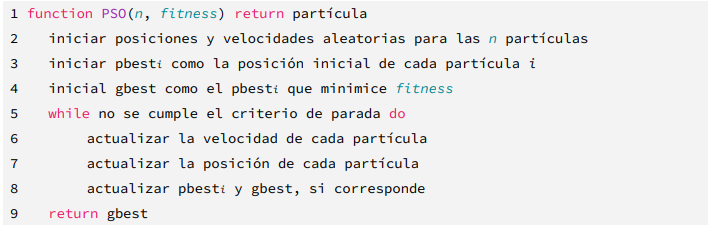
## Optimización por Enjambre de Partículas

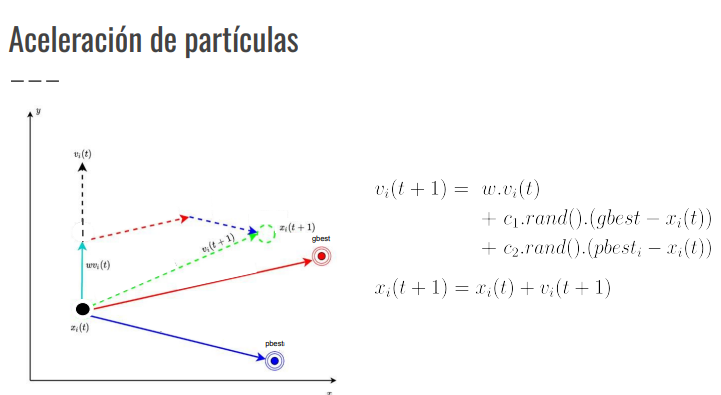
Particle Swarm Optimization (PSO)

La **optimización por enjambre de partículas** se inspira en el comportamiento de una bandada de aves, un cardumen de peces o un enjambre de insectos.

Un **enjambre** es una colección (población) aparentemente desorganizada de individuos en movimiento que tienden a agruparse, mientras cada individuo parece moverse en una dirección aleatoria.

* Se utiliza para minimizar/maximizar una **función de fitness** en espacios continuos, cuyo gradiente es desconocido.
* La población inicial se compone de un conjunto de **partículas** (puntos de R^n) que inicialmente tienen **posición** y **velocidad** aleatorias.
* Cada partícula i recuerda su mejor **posición personal** (pbesti) y la mejor **posición global** (gbest) según la función de fitness.
* En cada iteración, cada partícula se acelera estocásticamente hacia pbest y gbest.





En cada iteración, la **velocidad** de la partícula i se acelera sumando:

* Vector celeste. Su velocidad multiplicada por un factor de inercia.
* Vector rojo en línea de puntos. Vector que apunta a gbest multiplicado por un número aleatorio y una constante c1.
* Vector azul en línea de puntos. Vector que apunta a pbest multiplicado por un número aleatorio y una constante c2.

La posición de la partícula i se actualiza sumando su última posición con el vector verde en línea de puntos (nueva velocidad), es decir, la suma de todo lo anterior.