Búsqueda

Libro

- Aritificial Intelligence : A modern approach
  Stuart Rusell y Peter Norvig
  http://aima.cs.berkeley.edu/

Agentes

- Agentes
  - Perciben al ambiente a través de sensores
  - Actúan en el ambiente a través de actuadores
- Agentes racionales
  - Ejecuta la acción "correcta"
  - Por cada posible secuencia de percepciones, un agente racional debe elegir la acción que maximice su medida de desempeño.

Ambientes

- Completamente o parcialmente observable
- Determínistico o estocástico
- Episódico o secuencial
- Estático o dinámico
- Discreto o continuo
- Único o multi-agente

Partes del agente

- Función del agente
  - De percepción a acción
- Programa del agente
  - Implementación de la función del agente
- Arquitectura del agente
  - Cuerpo del agente
- Agente = Programa + Arquitectura
  Agente = Mente + Cuerpo

Tipos de agentes

- Agentes reactivos
  - Acción depende de la percepción y el estado actual
- Agentes basados en modelos
  - Estado actual depende de la historia de estados
- Agentes orientados a metas
  - Acción depende de que se trata de hacer
- Agentes basados en utilidad
  - Acción depende de que tan útil es para alcanzar el objetivo

Agente aspiradora

- Agentes reactivos
  - Aspiro si está sucio o no, se mueve aleatoriamente
- Agentes basados en modelos
  - Se mueve de izquierda-derecha, arriba-abajo
- Agentes orientados a metas
  - Tiene que limpiar de la sala a la cocina
- Agentes basados en utilidad
  - Gastar la menos batería posible

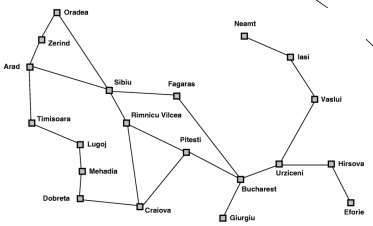
Agentes solucionadores de problemas

- Son agentes orientados a metas
- Deciden que secuencias de acciones hacer para alcanzar la meta
- Dos palabras:
  - Problema
  - Solución

Objetivos y estados

- Considere qué un agente puede estar en varios estados
  - Un subconjunto de los estados son deseables para el agente: Son la meta
  - La tarea para el agente es alcanzar esos estados a través de acciones.

Ejemplo: Viaje por Rumania

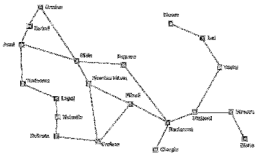


- Un agente está Arad
- Quiere llegar a Bucharest

Nivel de abstracción

- Estados:
  - Calles?
  - en(Ciudad)
- Objetivos:
  - en(Bucharest)
- Acciones:
  - Mover pie izquierdo?
  - ir(Ciudad)

Búsqueda



- en(Arad)
- ??
- ??
- ??
- ??
- en(Bucharest)

El proceso computacional de evaluar las diferentes acciones para llegar a una meta se le conoce como búsqueda

El agente solucionador

- Recibe un problema
- Regresa una solución
  - Secuencia de acciones
- De manera general utiliza búsqueda

Ambientes

- Completamente o parcialmente observable
- Determínistico o estocástico
- Episódico o secuencial
- Estático o dinámico
- Discreto o continuo
- Único o multi-agente

Warning: suposiciones ☹

- Completamente observable
- Determínistico
- Episódico
- Estático
- Discreto
- Único

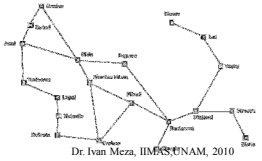
Es pura mente

Un problema

- Estado inicial
- Acciones
  - Sucesor
  - {(go(yaxchilan),In(yaxchilan)), (go(aguaAzul),in(aguaAzul))}
- Prueba de arribo a meta
  - ¿ya llegué?
- Función de costos:
  - Costo de llegar a la ciudad: tiempo, distancia

Espacio de búsqueda

- Estados + acciones definen el espacio de búsqueda
  - El conjunto de estados alcanzables desde el estado inicial

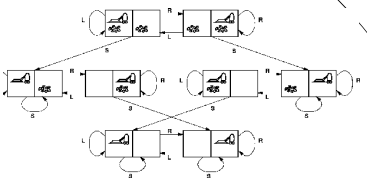


Solución óptima

- Una solución es una secuencia de estados conectados por acciones cuyo primer estado es el inicial y último estado es un estado meta
- Una solución óptima es la solución con el menor costo de todas las soluciones.

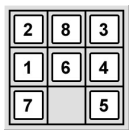
Problemas ejemplo

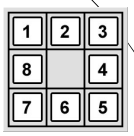
- Aspiradora
  - 8 estados



Problemas ejemplo

- 8-puzzle
  - 8:181,440; 15:1.4 trillón y 24:10^{25}

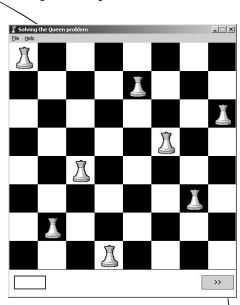




- NP-complete
  - Nadie espera encontrar algoritmo mejor que los de busque que aquí vamos a ver

Problemas ejemplo

- 8-reinas
  - Original: $1.8*10^{14}$
  - Reducido: 2057
- 100-reinas
  - Original: 10^{400}
  - Reducido: 10^{52}



Problemas reales

- Encontrar una ruta
- Traveling salesman problem
- Impresión de chips
- Navegación de robots
- Ensamblado de partes

Desempeño del agente

- Completes
  - Encuentra la solución cuando existe una
- Optimalidad
  - Encuentra la solución óptima
- Complejidad temporal
  - Cuanto toma encontrarla
- Complejidad espacial
  - Cuanta memoria necesita

Mediciones Sol. Probs.

- Espacio
  - En computación teórica: espacio de búsqueda
  - En IA:
    - *b* Factor de ramificación (max número de sucesores)
    - *d* Profundidad del primer estado meta
    - *m* Máxima longitud de una secuencia de soluciones
- Tiempo
  - Nodos generados

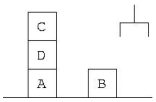
Técnicas de búsqueda

- ciega o sin información
  - No hay información extra sobre los estados
- con heurísticas o informada
  - Sabemos que estado es más "prometedor"

Búsquedas a ciegas

- Búsqueda con prioridad en achura
- Búsqueda con costo uniforme
- Búsqueda con prioridad en profundidad
- Búsqueda con profundidad limitada
- Búsqueda con profundidad iterativa
- Búsqueda bidireccional

Nuevo ejemplo



- Ontable(A)
- Ontable(B)
- On(D,A)
- On(C,D)
- Clear(C)
- Clear(B)
- Handempty

Prioridad en anchura

- Primero expandir el estado inicial
- Luego los sucesores
- Luego los sucesores de los sucesores
- Luego los sucesores de los sucesores …
- En cada expansión checamos si encontramos un estado meta

Implementación

```
def tree_search(problem, fringe):
  fringe.append(Node(problem.initial))
  while fringe:
     node = fringe.pop()
     if problem.goal_test(node.state):
        return node fringe.extend(node.expand(problem))
  return None
```


Implementación (2)

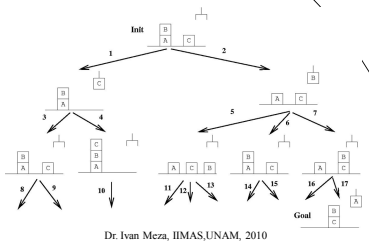
```
def expand(self, problem):
   return [
        Node(next,
                self,
                act,
                problem.path_cost(self.path_cost,
                        self.state,
                        act,
                        next)) for (act, next) in
                                problem.successor(self.state)]
```

Diferencia Nodo y Estado

- El estado define nuestro problema
- El nodo es una estructura de datos de la búsqueda:
  - Padre
  - Acción
  - Profundidad
  - Costo

Implementación (3)

- BPA = tree-search(problem,FIFO/Fila)

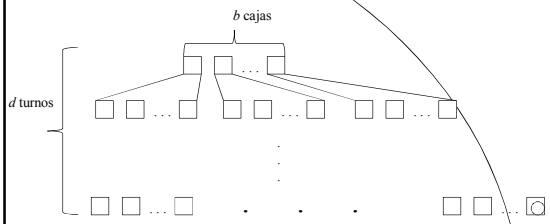


El agente malvado

- Un malvado agente guardará una canica roja dentro de una caja

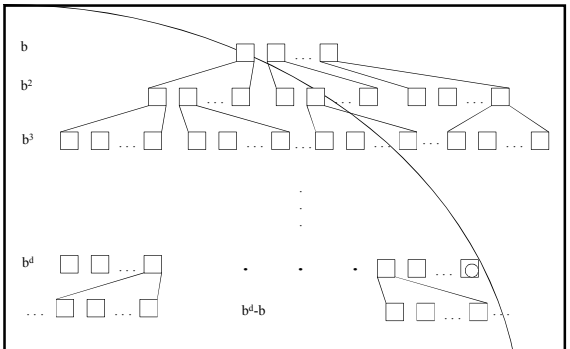
  - En cada turno, abrimos una caja que tiene *b* cajas o la canica roja
  - La canica tiene que estar dentro de *d* cajas
  - El número máximo de cajas a sacar es *m*
  - Podemos *buscar* en todas las cajas como queramos
- El agente malvado siempre esconde la canica en la última caja del último turno

El juego del agente malvado



Metáfora para búsqueda

- Las cajas son nodos
  - b es el factor de ramificación
  - d es la profundidad de la meta
  - m es la máxima profundidad de las soluciones
- ¿Qué haríamos nosotros?
- ¿Qué hacen las estrategias de búsqueda?
- El escenario del agente malvado supone el peor de los casos!



Evaluación BPA

- Sí *b* es finito y existe un *d* encontrará la solución
- Sin embargo considerar que el estado inicial tiene *b* sucesores
  - Los sucesores tendrán b^2
  - Los sucesores tendrán b^3
  - ... b^{d+1} -*b*
  - $O(b^{d+1})$

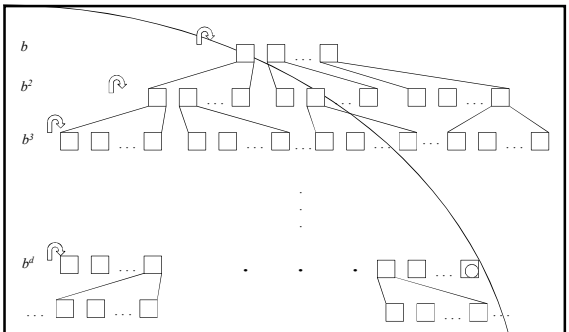
Ejemplo

Profundidad	Nodos	Tiempo	Memoria
2	1100	.11	1 MB
4	111,100	11	106 MB
6	10^7	19	10 GB
8	10^9	31	1 TB
10	10^{11}	129 días	101 TB
12	10^{13}	35 años	10 PB
14	10^{15}	3,523 años	1EB

b=10 y 10,000 nodos/seg y 1000 bytes/nodo

Búsqueda con costo uniforme

- Utilizar el costo para expandir!!
- Expandimos el nodo con el costo más bajo
- Si los costos son iguales se convierte en BPA



d=C*/ε

El agente malvado va escoger el costo en la cual guardar la canica y en el problema existe cada acción tiene almenos un costo de ε

Evaluación de BCU

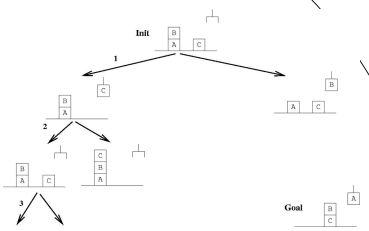
- Sí hay acciones con sin costo, entra en un loop infinito
- $O(b^{1+[C^*/\varepsilon]})$
- C* costo de la solución óptima
- ε costo mínimo de acción
  - Explora árboles de pasos con costos pequeños

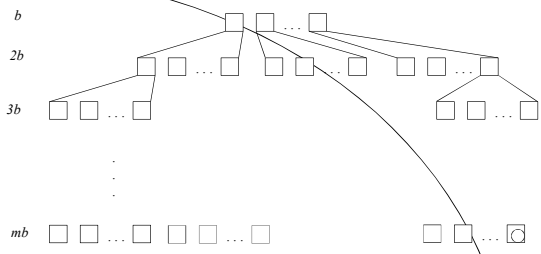
Búsqueda con prioridad en profundidad

- Primero expandir el estado inicial
- Luego el primer sucesor
- Luego el primer sucesor del primer sucesor
- Luego primer sucesor del primer sucesor …
- …
- Luego el segundo sucesor
- Luego el primer sucesor del segundo sucesor
- …

Implementación (3)

- BPP = tree-search(problem,LIFO/Pila)





Nodos creados: $b+b^2+b^3 \ldots b^m$

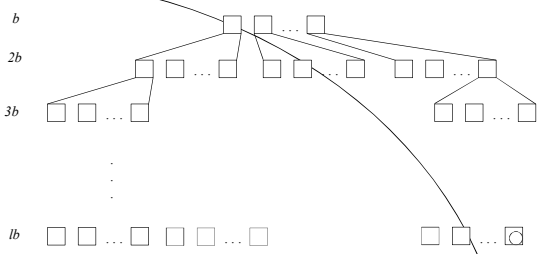
Agente suficientemente malvado para poner $d=m$

Evaluación BPP

- Espacio: *bm+1, O(bm)*
  - Esto reduce los 10PB a 118 KB
- Back-tracking: expandir únicamente un sucesor (e.g., prolog).
  - O(m)
- Sin embargo se puede quedar tardar en una rama (secuencia de acciones).
- Algunas veces se tarda un infinito

Búsqueda con profundidad limitada

- Expandir hasta cierta profundidad (*l*)
- Resolvemos el problema de ramas infinitas pero si:
  - $d > l$
- Algunas veces podemos garantizar que $d<=l$
- En esas ocasiones se le conoce como el diámetro



Nodos creados: $b+b^2+b^3 \ldots b^l$

Agente suficientemente malvado para poner $d=l$
¿Qué pasa si es más malvado y $d>l$?

Búsqueda con profundidad iterativa

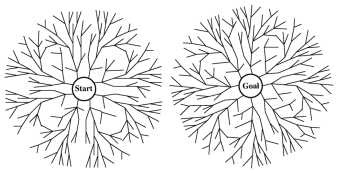
- Hago una búsqueda con BPL con l=1
- No funciono uso BPL con l=2
- No funciono uso BPL con l=3
- No funciono uso BPL con l=4
- …

Evaluando BPI

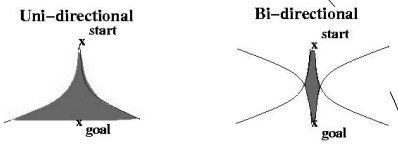
- ¿Tonto?
  - Repasamos los nodos que ya visitamos!
- Memoria requerida:
  - *O(bd)*
- Espacio requerido
  - Nodos=$(1)b^d+(2)b^{d-1}+\ldots+(d-1)b^2+db$
  - BPA= $b+b^2+\ldots+b^{d-1}+b^d+b^{d+1}-b$

Búsqueda bidireccional

- Sí comienzo por la meta!
- Si comienzo por el inicio y la meta!



Ilustración



Evaluando

- Sí usamos BPA en ambos lados
  - $O(b^{d/2})$!!
- Pero hay que definir como ir para atrás!
- Y si no sabemos la meta

Comparación

	BPA	BCU	BPP	BPL	BPI	BB
Completes	Sí (a)	Sí (a,b)	No	No	Sí (a)	Sí (a,d)
Tiempo	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^k)$	$O(b^{d/2})$
Espacio	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Óptimo	Sí (c)	Si	No	No	Sí (c)	Sí (c,d)

a : *b* finito
b: costo ≥ ε
c: costo idénticos
d: Usan BPA

Otras mejoras

- ¿Qué tal si llego a un estado que ya visité?
  - Comparar nuevos nodos con los viejos!
- ¿Buena idea?
  - En BPP evita loops
  - Sin embargo si el estado no está visible en la pila no se detecta
- *Algoritmos que olvidan su historia están condenados a repetirla*

De árbol a grafo

- Descartar los nodos ya creados
- En lugar de explorar un árbol se explora un grafo
  - Espacio y tiempo proporcional al espacio de búsqueda
- Optimalidad: tiene truco!
  - Dos nodos similares, dos caminos al mismo estado ¿cuál desechar?

Búsqueda con heurísticas

- Búsqueda por primer mejor
- Búsqueda codiciosa
- Búsqueda A*
- Búsqueda con profundidad iterativa y A*
- Búsqueda por primer-mejor recursivo

Búsqueda por primer-mejor

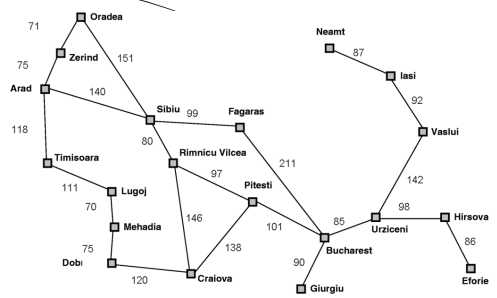
- Idea expandir únicamente el que tenga menor costo
  - Mejora el espacio
  - Pero sigue cayendo en ciclos infinitos

Heurística

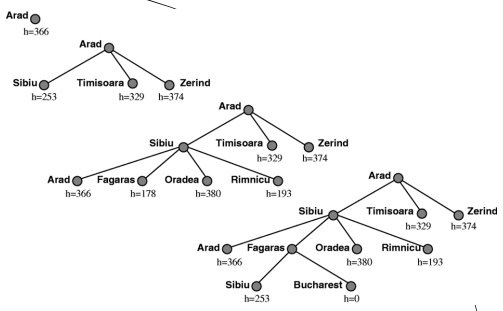
- Es una función que estima el costo de cualquier nodo a un nodo meta
  - Representa conocimiento sobre el problema
  - En el mapa Rumano distancias en línea recta entre ciudades

Búsqueda codiciosa

- Y si uso mi heurística que opción está más cerca de la meta !



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu	193
Fagaras	176	Sibiu	253
Giugiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

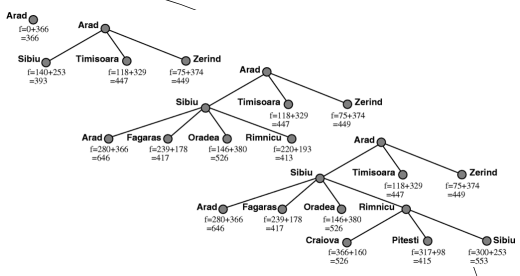


Evaluando

- No óptima
- No completo
- $O(b^m)$
- Todo depende de la calidad de la heurística

Búsqueda A*

- La función de costo es:
  - $f(n) = g(n) + h(n)$
- g es el costo de alcanzar n
- h es el costo de llegar del nodo al nodo meta
- f termina siendo el costo de pasar por el nodo



La función *h*

- A* es óptima si h no sobre estima el costo de llegar a la meta
  - h tiene que ser razonable
- Cuando usamos un grafo:
  - Descartar el camino con más costo
  - h tiene que se monotónica
    - $h(n) \leq c(n,a,n') + h(n')$

Más sobre A*

- Supongamos que C* es el costo de la función óptima
  - A* expandirá nodos con $f(n) < C^*$
  - Crea un contorno
- A* es completa ya que no expandirá nodos con $f(n) > C^*$
- Es eficiente!!

Las malas noticias

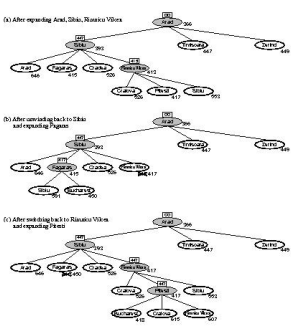
- El contorno es exponencial en muchos caso
  - A* necesita que el error de la heurística no crezca más rápido que el logaritmo de del costo real

Búsqueda con profundad iterativa y A*

- BPIA*
- En lugar de parar por profundidad paramos por
  - El costo menor de los costos que excedieron *f*
- Problemas con costos con valores reales

Búsqueda por primer-mejor recursivo

- Qué tal si uso BPM pero guardo el segundo mejor camino!
  - Si el costo actual es mayor que el segundo mejor, entonces cambio al segundo mejor camino
- Termina iterando entre primer y segundo camino!
- BPMR > BPIA*

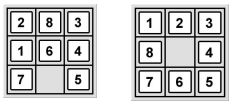


Búsqueda limitada por la memoria A*

- Similar a A* pero borra nodos, para evitar alcanzar no rebasar el límite de memoria
  - Expandir el mejor y más nuevo nodo
  - Borrar el peor y más viejo nodo
- Es completo si la solución es alcanzable con la memoria disponible
- Es optimo de igual manera
- Tiende a cambiar entre nodos borrados y nuevos

Heurísticas

- 8-puzzle



  - 22 pasos
  - 3.1×10^{10} y con estados repetidos 170,000

Dos heurísticas

- h1: Número de cuadros en un lugar incorrecto
- h2: Distancia de los cuadros a su posición original

Desempeño 1200 problemas (2-24)

d	IDS	A*(h1)	A*(h2)
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14	-	539	113
16	-	1301	211
18	-	3056	363
20	-	7276	676
22	-	18094	1219
24	-	39135	1641

Relajar el problema

- 8-puzzle moviéndose a cualquier lado
  - Un heurística en 8-puzzle relajado es válida en 8-puzzle
- Método para descubrir heurísticas
  - Crear versiones relajadas del problema
  - Crear heurísticas de esas versiones relajadas
  - Resolver problema original con todas heurística

Y si aprendemos la heurística

- Resolver muchos 8-puzzle
- Cada solución es ejemplo de la solución y el costo
  - Redes neuronales, árboles de decisión, etc
- Características
  - Ejem. Número de cuadros erróneos (si 5 h es 14)
  - $h(n) = c_1x_1(n)+c_2x_2(n)$

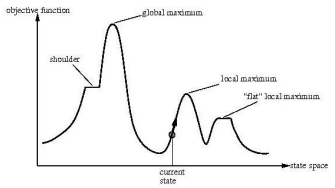
Otra dirección…

- Y qué tal si nos olvidamos de las secuencias de soluciones!
  - Sí utilizamos únicamente un estado global
    - Poca memoria
    - Encuentran soluciones razonables a los problemas
  - Problemas de optimización con una función objetivo
  - Encontrar el mejor estado usando la función objetivo

Búsqueda local!

- Observo si alguna acción me acerca a la solución,
  - Camino hacia ese estado
  - Deambulo usando esa estrategía

Espacio de vista



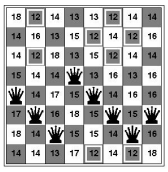
Escalada por pendiente

```
def hill_climbing(problem):
    current = Node(problem.initial)
    while True:
        neighbor = argmax(expand(node, problem), Node.value)
        if neighbor.value() <= current.value():
                return current.state
        current = neighbor
```


8-reinas

- Un estado: 8 reinas en el tablero
- Función sucesor: posibles tableros al mover una reina
- Función de costo: Número de reinas que se atacan entre sí (directa o indirectamente)
  - Escoger aleatoriamente de las opciones posibles

Sucesores de 8-reinas



- Búsqueda local codiciosa!

Codicia

- Pecado capital
- No tan malo para nosotros, pero
  - Máximos locales
  - Monturas
  - Planos
- Obtiene una solución: 14%
- Sólo toma 4 pasos encontrar la solución, y 3 no encontrarla

Mejorar la solución

- El programa para cuando no hay un mejor estado, estamos en un plano
- Y si permitimos movernos a uno de estos estados (sideways), solamente un número de veces
  - De 14% a 94%
  - Sin embargo, las soluciones en promedio le toma 21 pasos encontrar la solución y 64 darse por vensido

Otras opciones

- Escalada de pendiente estocástica
- Primera opción de escalada de pendiente
- Reiniciar aleatoriamente la escalada de pendiente
  - 7 reinicios, 22 pasos
  - Sideways: 1.06 reinicios, 25 paso
  - 3 millones de reinas en menos de un minuto

Problemas NP-hard

- Si el perfil tiene pocos máximos locales y planos, fácil para escalada de pendiente
- Pero, NP-hard luce como una familia de puercoespines viviendo en un piso plano

Escalada + temblor

- Algoritmos de caminata aleatoria
  - Una bola de ping-pong en un mesa con valles y colinas
  - Tirar la pelota dejarla y mover la mesa
  - Conforme avanza, disminuir la velocidad de mover la pelota
- En práctica el cambio aleatorio se acepta o se rechaza

Búsqueda local con *beam*

- En lugar de tener un estado, tener *k* estados
  - Los estados no son tan variados

Algoritmos genéticos

- Los estados sucesores están creados de la unión de dos estados padres
  - Selección natural
  - Comienza con *k* estados aleatorios
- Función de aptitud ~ función de costo
  - Selcción
  - Cruza
  - Mutación

Ilustración: AG

Evaluación de AG

- Semejante a escalada de pendiente con caminata aleatoria
- Ventajas en la cruza
  - Si la cruza toma elemento aleatorios AG pierden poder de búsqueda
- Esquemas de solución prevalecen
- Es necesario representar la solución en términos del problema real (Ingeniería de la representación)