



Inteligencia Artificial

Agentes que resuelven problemas

1. Formulación de meta (decidir que estados son objetivo) y del problema (decidir que acciones y estados se van a considerar)
2. Buscar una solución (examinar posibles acciones y armar una secuencia)
3. Ejecutar la solución

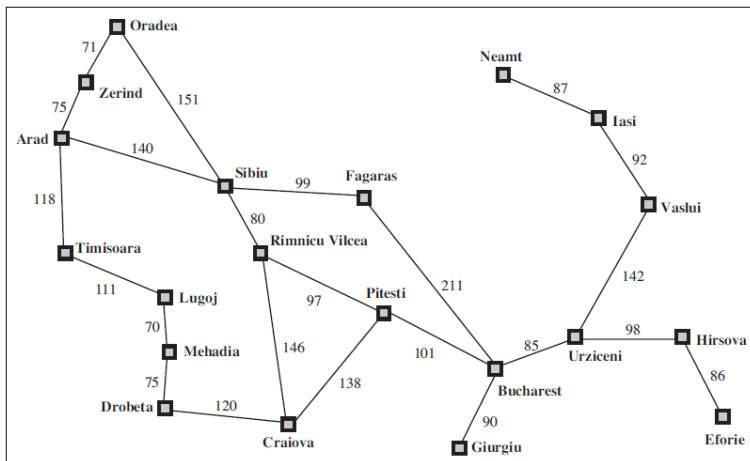
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
               state, some description of the current world state
               goal, a goal, initially null
               problem, a problem formulation

  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
    if seq = failure then return a null action
  action  $\leftarrow$  FIRST(seq)
  seq  $\leftarrow$  REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Problemas y soluciones bien definidas

El mapa de Rumania.



Un problema puede ser definido formalmente mediante **5 componentes**:

- **Estado inicial**
- **Acciones** -> función *Actions(s)*
- **Modelo de transición** -> función *Result(s, a)*
- **Comprobación de meta**
- **Costo de camino o costo de paso**

Una **solución** es una secuencia de acciones que, aplicadas al estado inicial, me dejan en un estado meta.

Una **solución óptima** es una solución donde el costo de ruta es el **mínimo** de todas las soluciones posibles.

Formulando problemas

Para formular los problemas es obligatorio hacer **abstracciones**

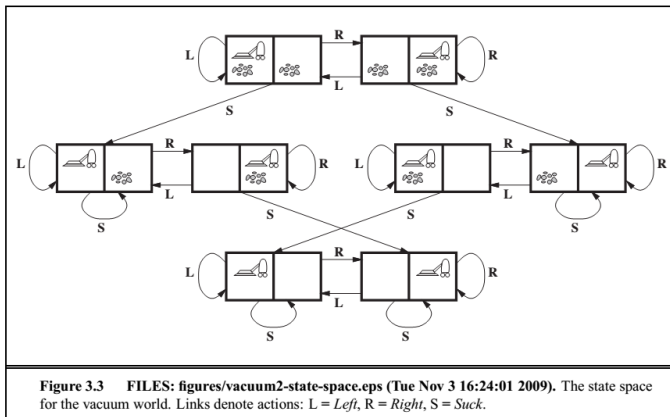
¿qué abstraer?

- **Estados:** debe remover todos los detalles posibles conservando lo mínimo necesario para cumplir nuestro objetivo
- **Acciones:** contemplar solo las acciones necesarias, expresadas de la manera más sencilla posible

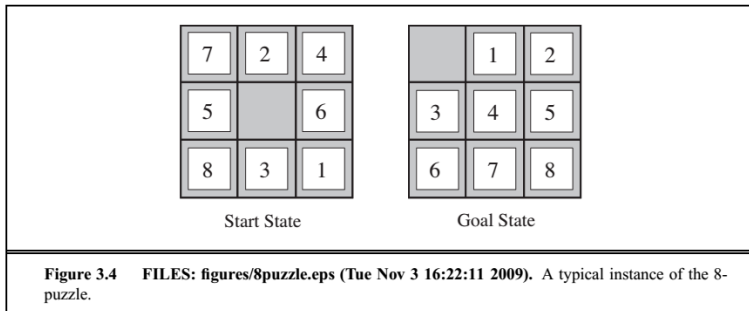
Problemas de ejemplo

Problemas de juguete:

El mundo de la aspiradora



8-puzzle



8 reinas

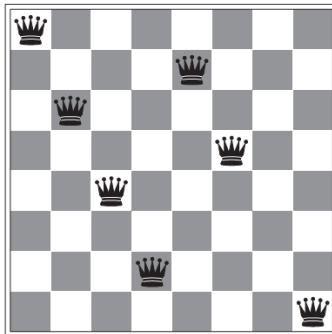


Figure 3.5 FILES: figures/8queens.eps (Wed Nov 4 16:21:52 2009). Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

Problemas reales:

- Búsqueda de rutas
- El problema del turista
- Navegación de robots

Buscando soluciones.

- **Arbol de búsqueda:** las aristas son acciones y los nodos se relacionan a estados

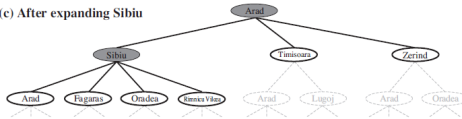
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



- **Expandir** el estado actual significa aplicar las acciones posibles y generar un nuevo conjunto de nodos
- **Frontera** o **lista abierta** es la lista de nodos **hoja** que todavía no fueron *procesados*
- **Estrategia de búsqueda** determina que nodo debe expandirse
- **Conjunto de nodos explorados** o **lista cerrada** son todos aquellos estados que fueron procesados
- El algoritmo que tiene en cuenta la lista cerrada se denomina **búsqueda en grafo**; mientras que el que **no** la tiene en cuenta, **búsqueda en árbol**.

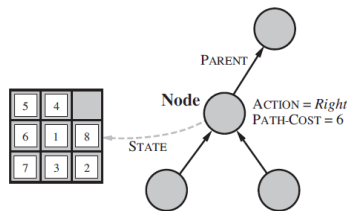
```
function TREE-SEARCH(problem) returns a solution, or failure  
  initialize the frontier using the initial state of problem  
  loop do  
    if the frontier is empty then return failure  
    choose a leaf node and remove it from the frontier  
    if the node contains a goal state then return the corresponding solution  
    expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure  
  initialize the frontier using the initial state of problem  
  initialize the explored set to be empty  
  loop do  
    if the frontier is empty then return failure  
    choose a leaf node and remove it from the frontier  
    if the node contains a goal state then return the corresponding solution  
    add the node to the explored set  
    expand the chosen node, adding the resulting nodes to the frontier  
      only if not in the frontier or explored set
```

Infraestructura para algoritmos de búsqueda

La estructura de un nodo se define mediante:

- Estado
- Nodo padre
- Accion
- Costo de camino



La estructura para la **frontera** es una lista que se comporta como **Cola**, **Pila** o **cola priorizada**

El conjunto de estados explorados puede ser implementado como una tabla hash.

Midiendo performance de los algoritmos

- **Complejidad:** el algoritmo asegura encontrar una solución, si la hay
- **Optimalidad:** el algoritmo encuentra una **solución óptima**
- **Complejidad temporal:** ¿Cuánto tiempo toma encontrar la solución?
- **Complejidad espacial:** ¿Cuánta memoria necesito para encontrar la solución?

Las medidas de complejidad se van a expresar en términos de:

- **b** el factor de ramificación
- **d** la menor profundidad de algún nodo meta
- **m** la longitud máxima de camino en el grafo de estados

El tiempo se mide en términos de la cantidad de nodos generados; mientras que el espacio según la cantidad de nodos en memoria.

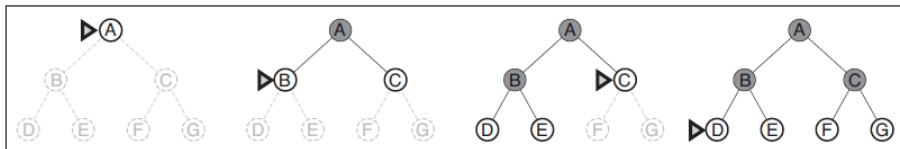
Vamos a usar el **costo de búsqueda** en lugar del **costo total**.

Estrategias de búsqueda sin información

- Búsqueda en amplitud
- Búsqueda de costo uniforme
- Búsqueda en profundidad
- Búsqueda en profundidad limitada
- Búsqueda en profundidad iterativa
- Búsqueda bidireccional

Búsqueda en amplitud

Utiliza una **cola** en la frontera.



El número total de nodos generados es: $b + b^2 + b^3 + \dots + b^d = O(b^d)$

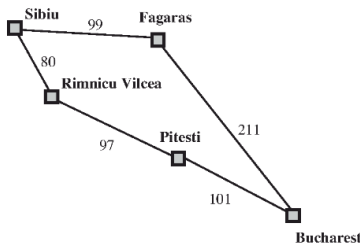
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure  
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0  
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)  
  frontier  $\leftarrow$  a FIFO queue with node as the only element  
  explored  $\leftarrow$  an empty set  
  loop do  
    if EMPTY?(frontier) then return failure  
    node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */  
    add node.STATE to explored  
    for each action in problem.ACTIONS(node.STATE) do  
      child  $\leftarrow$  CHILD-NODE(problem, node, action)  
      if child.STATE is not in explored or frontier then  
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)  
        frontier  $\leftarrow$  INSERT(child, frontier)
```

Búsqueda de Costo uniforme

Utiliza una **cola priorizada** en la frontera, ordenada por el costo de camino. La función $g(n)$ devuelve el costo del camino desde el nodo inicial al nodo actual.

Dos diferencias en el algoritmo:

- la comprobación de meta se realiza recién cuando se va a expandir el nodo
- se agrega una chequeo por si se encontró un estado ya generado pero de menor costo



function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

node \leftarrow a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

frontier \leftarrow a priority queue ordered by PATH-COST, with *node* as the only element

explored \leftarrow an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node \leftarrow POP(*frontier*) /* chooses the lowest-cost node in *frontier* */

if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

 add *node*.STATE to *explored*

for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

child \leftarrow CHILD-NODE(*problem*, *node*, *action*)

if *child*.STATE is not in *explored* or *frontier* **then**

frontier \leftarrow INSERT(*child*, *frontier*)

else if *child*.STATE is in *frontier* with higher PATH-COST **then**

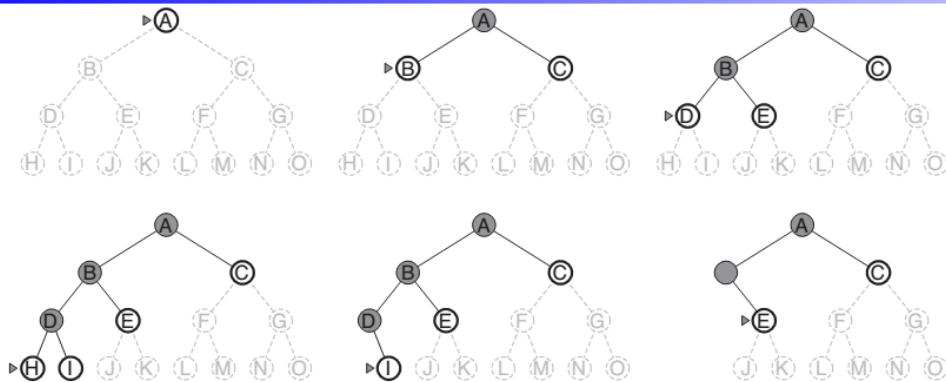
 replace that *frontier* node with *child*

Búsqueda en profundidad

Utiliza una **pila** en la frontera.

- La completitud depende de si hacemos búsqueda en árbol o en grafo
- No es óptima
- La complejidad temporal es $O(b^m)$ (puede ser peor que $O(b^d)$)
- La complejidad espacial es $O(b \cdot m)$

La variante **backtracking search** utiliza aún menos memoria.



Búsqueda en profundidad limitada

Igual a la búsqueda en profundidad, con la salvedad que se limita la profundidad máxima a alcanzar.

- Introduce otro problema para la completitud, el límite puede ser inferior a la profundidad de la mejor solución.
- Si conocemos el diámetro del espacio de estados, éste se puede usar como límite.

Búsqueda en profundidad iterativa

Encuentra el mejor límite aumentándolo gradualmente.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure  
  for depth = 0 to  $\infty$  do  
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)  
    if result  $\neq$  cutoff then return result
```

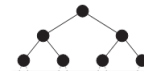
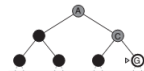
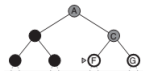
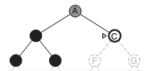
Limit = 0



Limit = 1

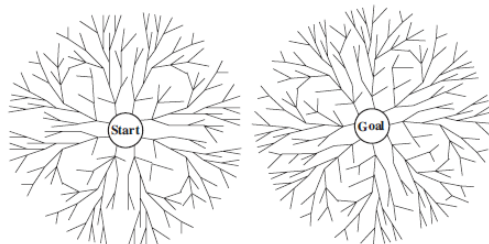


Limit = 2



Búsqueda bidireccional

- 2 búsquedas simultáneas, una desde el estado inicial y otra desde el estado meta
- $b(d/2) + b(d/2) \ll b^d$
- hay que tener acciones reversibles
- hay que conocer el estado meta



Resumen de los algoritmos

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Ejercicio.

1. Plantear formalmente el problema del 8-puzzle

1	4	2		1	2
	3	5	3	4	5
6	7	8	6	7	8

2. Resolver mediante búsqueda en grafo por:

- Amplitud
- Profundidad

3. Implemente en Python un agente que resuelva este problema.

Estrategias de búsqueda informada

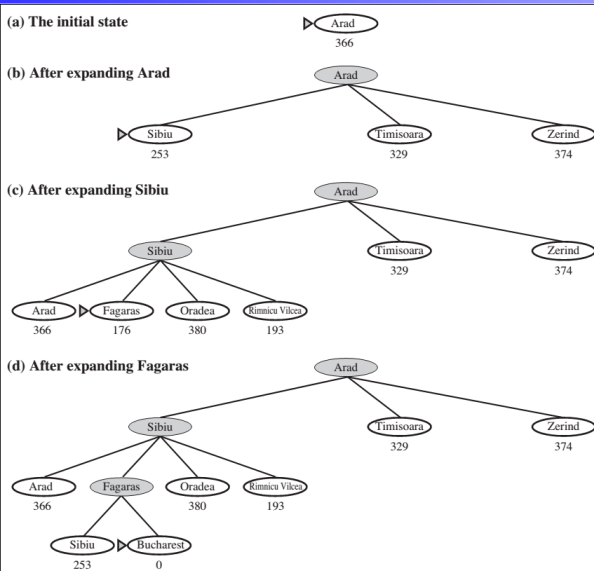
- Utilizan conocimiento específico del problema
- **best-first search** utiliza una función $f(n)$ para estimar costos
- $h(n)$ es una función heurística que generalmente forma parte de $f(n)$
- $h(n) = 0 \iff n$ es un nodo meta

Vamos a ver dos estrategias:

- Greedy best-first search (búsqueda avara)
- A^*

Búsqueda avara

- Expande el nodo que más se acerca a la meta
- $f(n) = h(n)$
- Si usamos búsqueda en árbol puede ser incompleta; si usamos búsqueda en grafo es completa en espacios no infinitos
- No es óptima
- La complejidad temporal y espacial en el peor de los casos es $O(b^m)$; pero esto depende mucho de la heurística
- Usa el mismo algoritmo que la búsqueda de costo uniforme, reemplazando $g(n)$ por $h(n)$

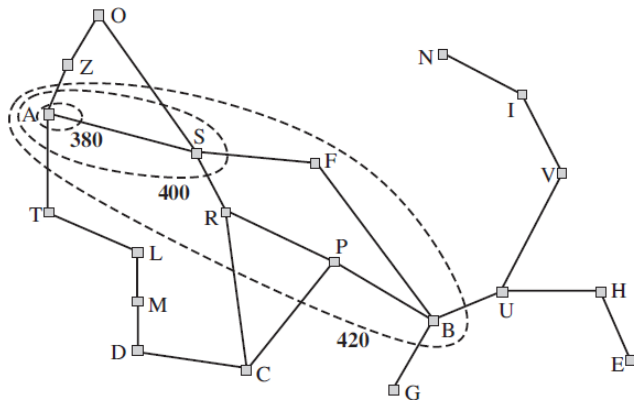


Búsqueda A* (A estrella)

- Minimiza el costo total estimado de la solución
- $f(n) = h(n) + g(n)$
- Usa el mismo algoritmo que la búsqueda de costo uniforme, reemplazando $g(n)$ por $g(n) + h(n)$
- Si $h(n)$ satisface determinadas condiciones, A* es completa y óptima.
 - **Admisibilidad:** Nunca sobreestima el costo de llegar a la meta
 - **Consistencia** (necesaria solo para búsqueda en grafo): $h(n) \leq c(n, a, n') + h(n')$

Optimalidad de A*

- A* busca en contornos de nodos del mismo costo total



- Si C^* es el costo de la solución óptima:

- A^* puede expandir algunos nodos dentro del contorno de la solución
- A^* expande todos los nodos con $f(n) < C^*$

- A^* es **óptimamente eficiente**: no hay otro algoritmo que garantice expandir menos nodos que A^* ; asegurando la optimalidad de la solución

Funciones heurísticas

Vamos a tomar de ejemplo 2 funciones heurísticas para el 8-puzzle:

- h_1 : número de fichas mal ubicadas
- h_2 : suma de las distancias de las posiciones de las casillas respecto a la posición de la solución. (midiendo mediante la distancia de Manhattan)

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

Una manera de caracterizar la calidad de una heurística es a través del **factor de ramificación efectivo** b^*

- $N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$

Una heurística bien diseñada tiene b^* cercano a 1.

Si para cada nodo n , $h_2(n) \geq h_1(n)$ decimos que h_2 **domina** a h_1 .

Generando heurísticas

- **desde problemas relajados:** el costo de una solución óptima de un problema relajado es una heurística admisible para el problema original
- Si se encuentran varias heurísticas alternativas, podemos generar una que domine a todas de la siguiente manera:

$$h(n) = \max\{ h_1(n), \dots h_m(n) \}$$

- **desde subproblemas:** el costo de solucionar un subproblema es menor que el de solucionar el problema completo

En las **bases de patrones** se almacenan los costos exactos para varios subproblemas, costos que luego son usados para calcular las heurísticas.

Ejercicio.

Implemente 1 heurística para el problema del 8-puzzle y compruebe el funcionamiento de A*.

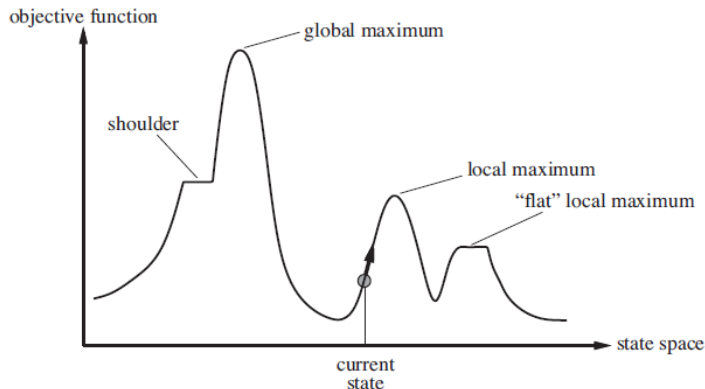
Más alla de la búsqueda clásica

Vamos a relajar las restricciones hechas previamente, donde los ambientes tenían que ser observables, determinísticos y conocidos; y las soluciones eran secuencias de acciones

- Búsqueda local
- Búsqueda con acciones no deterministas
- Búsqueda en ambientes parcialmente observables
- Búsqueda online

Búsqueda local

- Se usan cuando no importa el camino para llegar a la solución
- Mantienen solo un nodo en memoria y se mueven a sus vecinos (**usan poca memoria**)
- Son útiles en problemas de **optimización**
- Generalmente usan una formulación de estado completa.
- Una **función objetivo** determina que tan bueno es un estado. Esta función la podemos dibujar respecto al espacio de estados para obtener un paisaje del espacio de búsqueda.



- Un algoritmo es **completo** si siempre encuentra una solución, es **óptimo** si encuentra un máximo o mínimo global.

Ascenso de colina (Hill climbing)

- Siempre se mueve hacia arriba y termina cuando llega a un pico
- Tiene problemas con **máximos locales**, **crestas** y **mesetas**

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

current \leftarrow MAKE-NODE(*problem*.INITIAL-STATE)

loop do

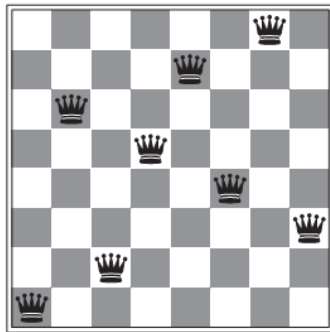
neighbor \leftarrow a highest-valued successor of *current*

if neighbor.VALUE \leq current.VALUE **then return** *current*.STATE

current \leftarrow *neighbor*

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♙	13	16	13	16
♙	14	17	15	♙	14	16	16
17	♙	16	18	15	♙	15	♙
18	14	♙	15	15	14	♙	16
14	14	13	17	12	14	12	18

(a)



(b)

Figure 4.3 FILES: figures/8queens-successors.eps (Wed Nov 4 16:23:55 2009) figures/8queens-local-minimum.eps (Wed Nov 4 16:14:15 2009). (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

Variantes.

- **Stochastic hill climbing:** elige al azar entre los movimientos ascendentes
- **First-choice hill climbing:** genera los sucesores aleatoriamente de a uno, hasta que uno sea mejor que el actual (es útil cuando el factor de ramificación es alto)
- **Random restart hill climbing:** ejecuta varias iteraciones de hill climbing tradicional, comenzando aleatoriamente desde distintos estados iniciales

Temple simulado (Simulated annealing)

- Combina el ascenso rápido con la aleatoriedad
- A medida que transcurre el tiempo (o disminuye la temperatura) la posibilidad de elegir un sucesor peor disminuye

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state

inputs: *problem*, a problem

schedule, a mapping from time to “temperature”

current \leftarrow MAKE-NODE(*problem*.INITIAL-STATE)

for $t = 1$ **to** ∞ **do**

$T \leftarrow$ *schedule*(t)

if $T = 0$ **then return** *current*

next \leftarrow a randomly selected successor of *current*

$\Delta E \leftarrow$ *next*.VALUE – *current*.VALUE

if $\Delta E > 0$ **then** *current* \leftarrow *next*

else *current* \leftarrow *next* only with probability $e^{\Delta E/T}$

Busqueda de Haz local

- Mantiene k nodos en memoria, en principio generados aleatoriamente
- Por cada paso se generan todos los sucesores y se eligen lo k mejores
- La información importante es compartida entre las distintas búsquedas paralelas
- Los k estados pueden concentrarse rápidamente en un sector pequeño del espacio de estados
 - Aparece una variante llamada **stochastic beam search**: toma aleatoriamente los k sucesores, de acuerdo al valor de la función de cada uno

Algoritmos genéticos

- Es una variante de la búsqueda de haz local, solo que los sucesores se generan de a pares
- Comienza con un conjunto llamado la **población**, compuesto de k estados o **individuos**
- Se usa una función **fitness**, la cual devuelve valores mayores cuando mejor es el individuo
- Se eligen $k / 2$ pares de individuos para ser reproducidos. La elección es al azar, con probabilidades proporcionales al fitness de cada individuo
- Los pares son sometidos a una operación de *crossover*
- Finalmente se pueden realizar mutaciones en los individuos resultantes

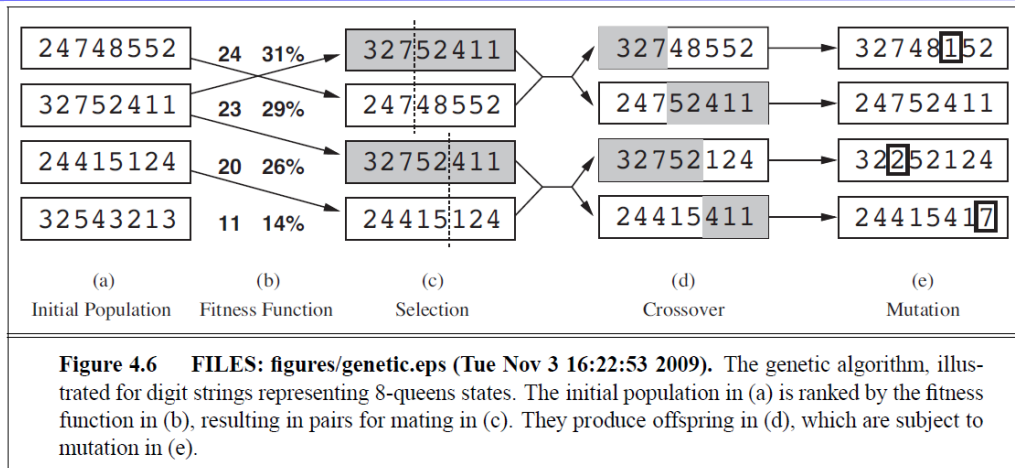


Figure 4.6 FILES: figures/genetic.eps (Tue Nov 3 16:22:53 2009). The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Más búsqueda...

- **Búsqueda con acciones no deterministas:** las soluciones devuelven **planes de contingencia** en lugar de secuencias de acciones
- **Búsqueda en ambientes parcialmente observables:** los algoritmos trabajan con **estados de creencia** en lugar de estados reales
- **Búsqueda online:** cuando el mundo o las acciones no son conocidos se necesita intercalar la búsqueda con la ejecución

Ejercicio.

Resuelva el problema de las 8-reinas mediante:

- hill-climbing con reinicio aleatorio
- temple simulado
- algoritmos genéticos.

Bibliografía y enlaces útiles.

- Russell S., Norvig P.: Artificial Intelligence: A modern Approach. Third Edition