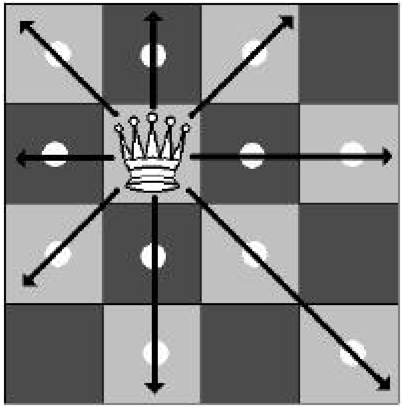
Inteligencia Artificial

Búsqueda

El Problema de las Ocho Reinas

- El problema de las ocho reinas consiste en poner ocho reinas en un tablero de ajedrez sin que se amenacen.
- Los movimientos posibles de una reina en el tablero son ilustrados en la siguiente figura:



Source: Wikipedia

El Problema de las Ocho Reinas

- El problema fue propuesto por Max Bezzel en 1848.
- El problema ha sido generalizado a las n-reinas.
- Como cada reina puede amenazar a todas las reinas que estén en la misma fila, cada una debe situarse en una fila diferente.

El Problema de las Ocho Reinas

- El vector de soluciones puede representarse como la posición en que se encuentra (columna) la reina en la fila i=1..8.
- El vector (3,1,6,2,8,6,4,7) significa que la reina 1 está en la fila 1 en la columna 3. La reina dos está en la fila dos y en la columna 1.
- ¿Qué pasa con las diagonales?

El Problema de las Ocho Reinas

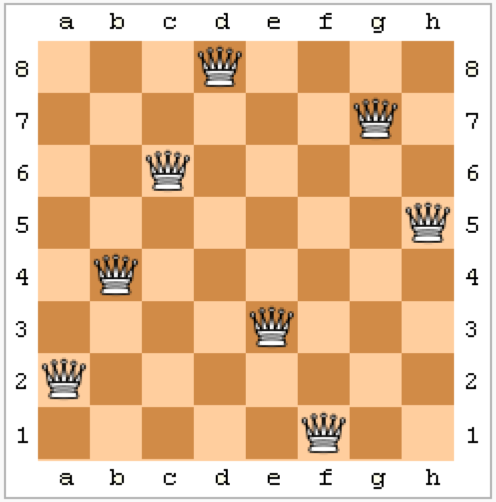
- Si tenemos dos reinas en posiciones (i,j) y (k,l) entonces están en la misma diagonal si se cumple:
  - $i-j = k-l$ o $i+j = k+l$
  - $j-l = i-k$ o $j-l = k-i$
- Por ejemplo, (3,5) y (6,8)
  - $3 - 5 = 6 - 8 \Rightarrow -2 = -2$
  - $5 - 8 = 3 - 6 \Rightarrow -3 = -3$

El Problema de las Ocho Reinas

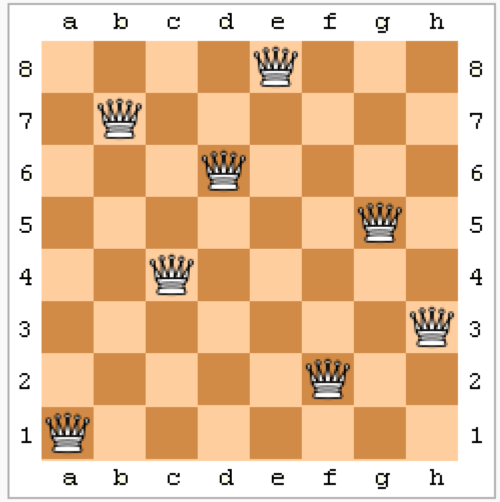
n	1	2	3	4	5	6	7	8	9	10	11	12
única	1	0	0	1	2	1	6	12	46	92	341	1787
distinta	1	0	0	2	10	4	40	92	352	724	2680	14200

n	13	14	24	26
única	9233	45752	28439272956934	2789712466510289
distinta	73712	365596	227514171973736	22317699616364044 87

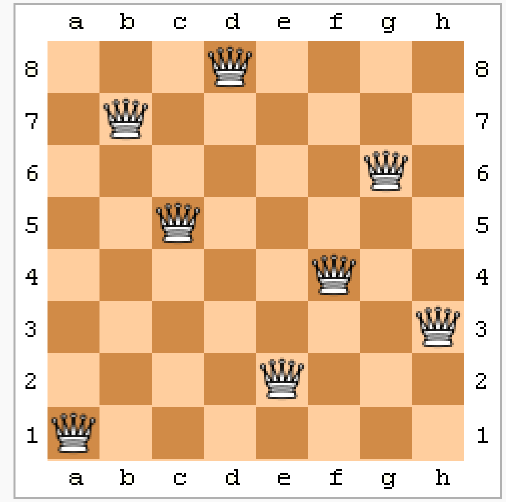
El Problema de las Ocho Reinas



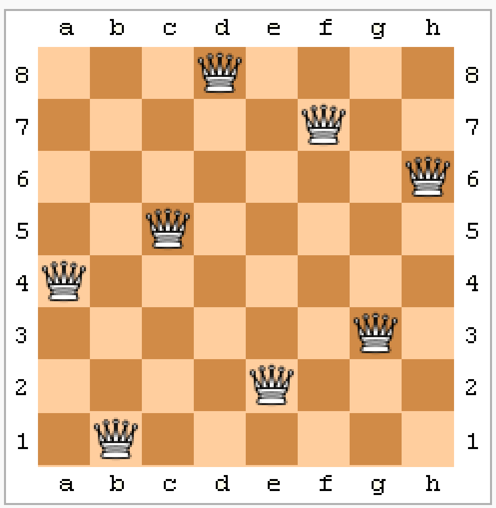
Solución única 1



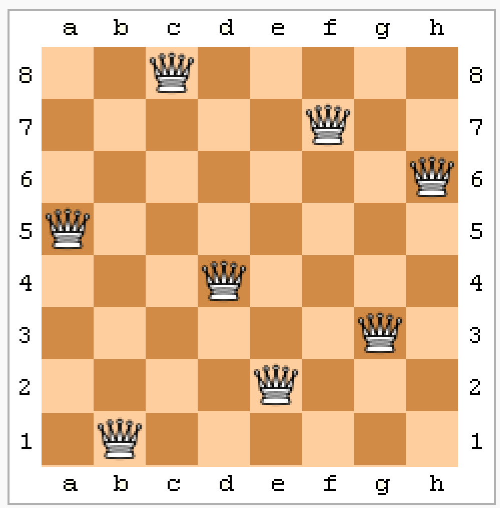
Solución única 2



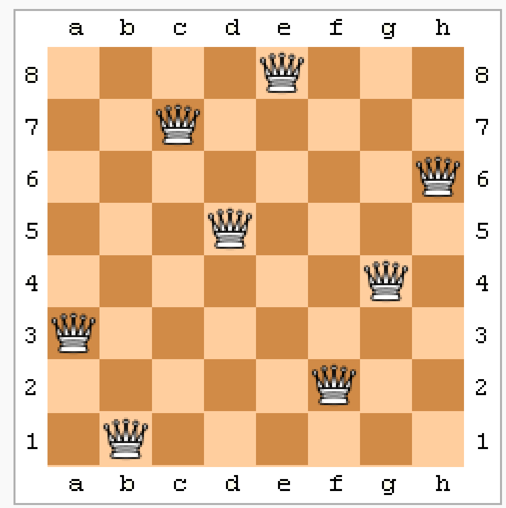
Solución única 3



Solución única 4

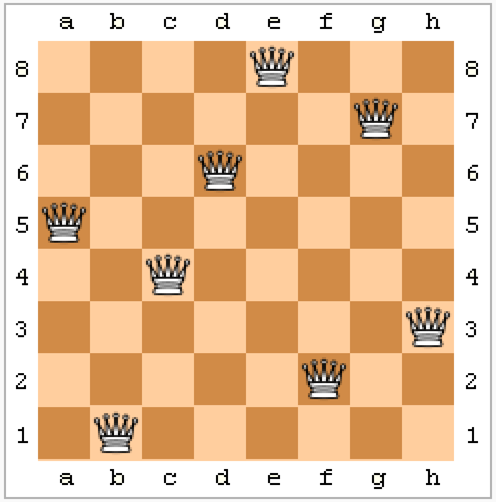


Solución única 5

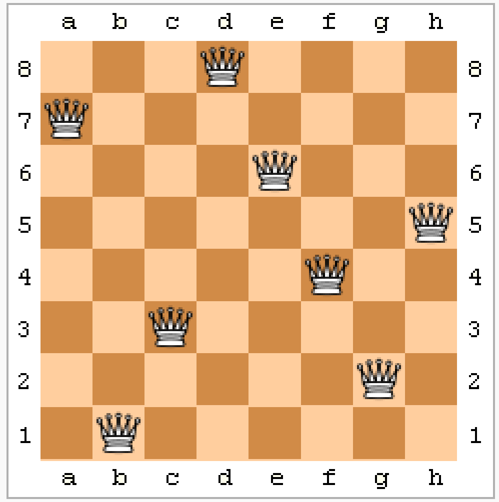


Solución única 6

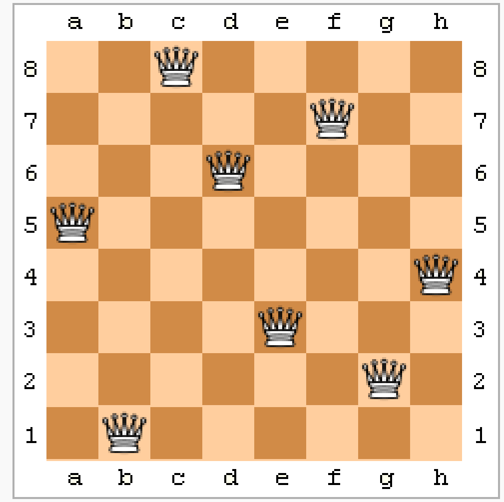
El Problema de las Ocho Reinas



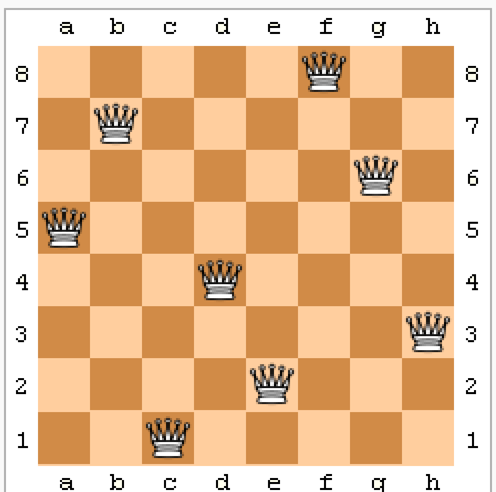
Solución única 7



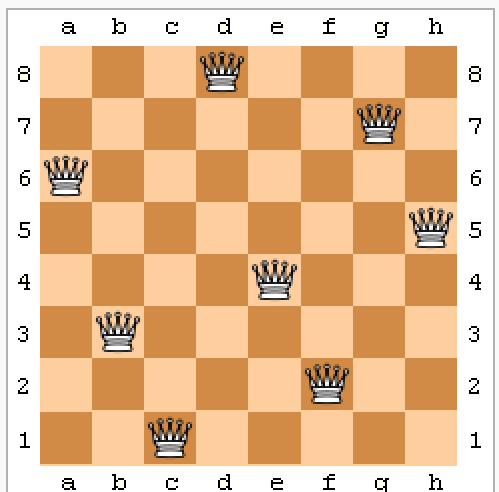
Solución única 8



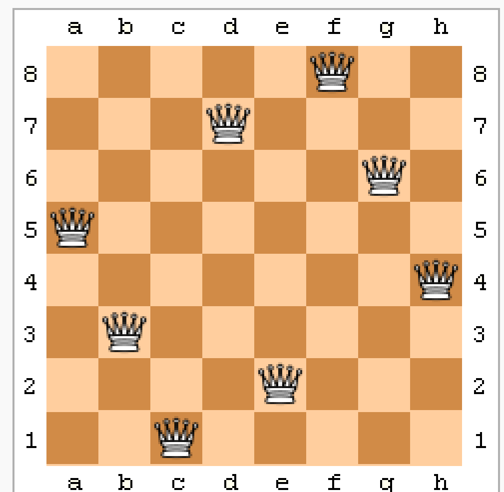
Solución única 9



Solución única 10



Solución única 11

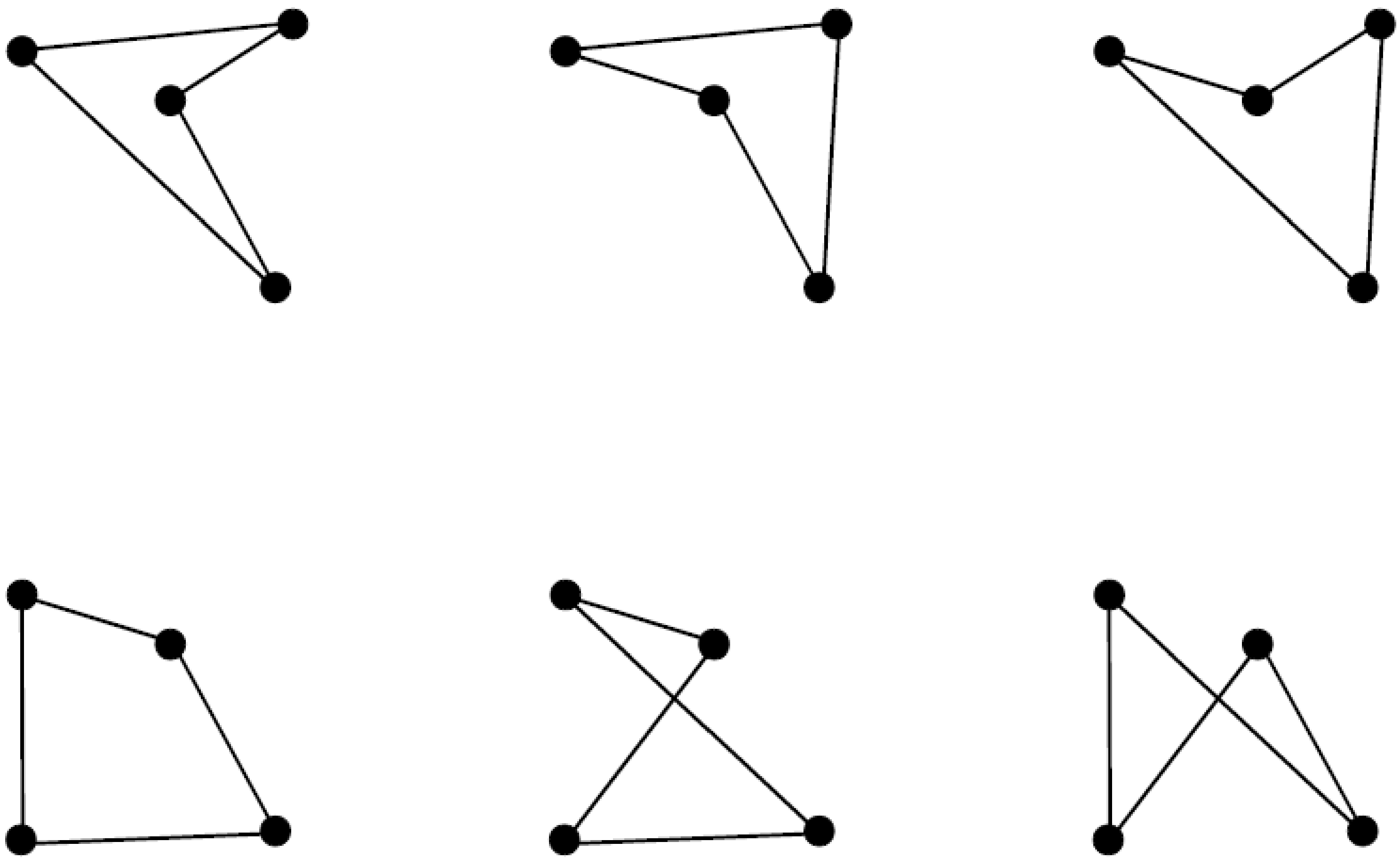


Solución única 12

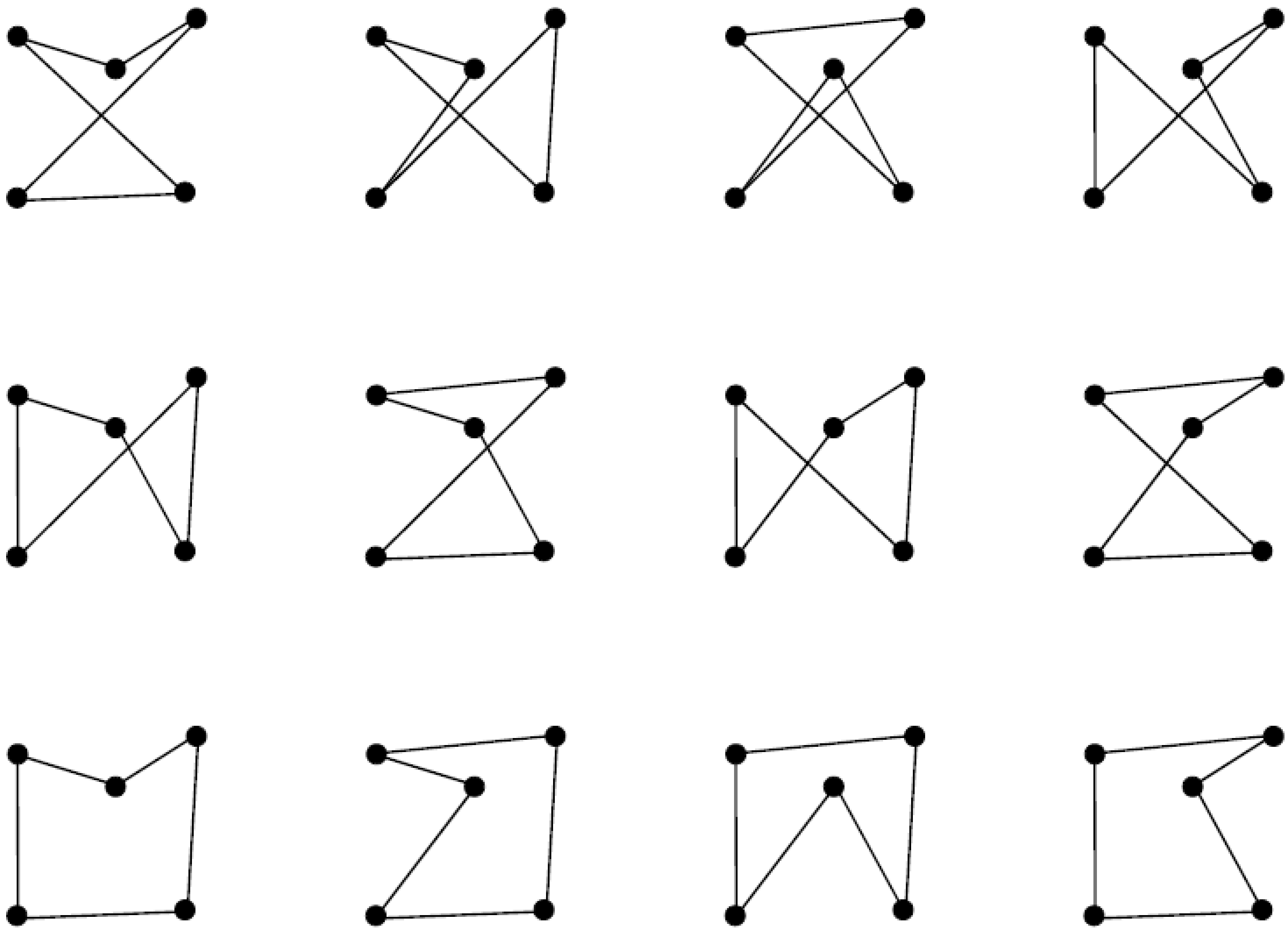
El Problema del Vendedor Viajero

- El escenario consiste en un vendedor que debe visitar un grupo de k ciudades:
  - Pasando sólo una vez por cada ciudad.
  - Volviendo a la ciudad de origen.
- ¿Cuál es el mejor recorrido?
  - El más corto
  - El más rápido
  - El más barato en términos de bencina.

El Problema del Vendedor Viajero



El Problema del Vendedor Viajero



El Problema del Vendedor Viajero

- ¿Cuántos recorridos hay en un problema de 10 ciudades?
  - 181440
- ¿Y en 50?
  - 304140932017133780436126081660647688443776415689605120000000000
- En general, con k ciudades:

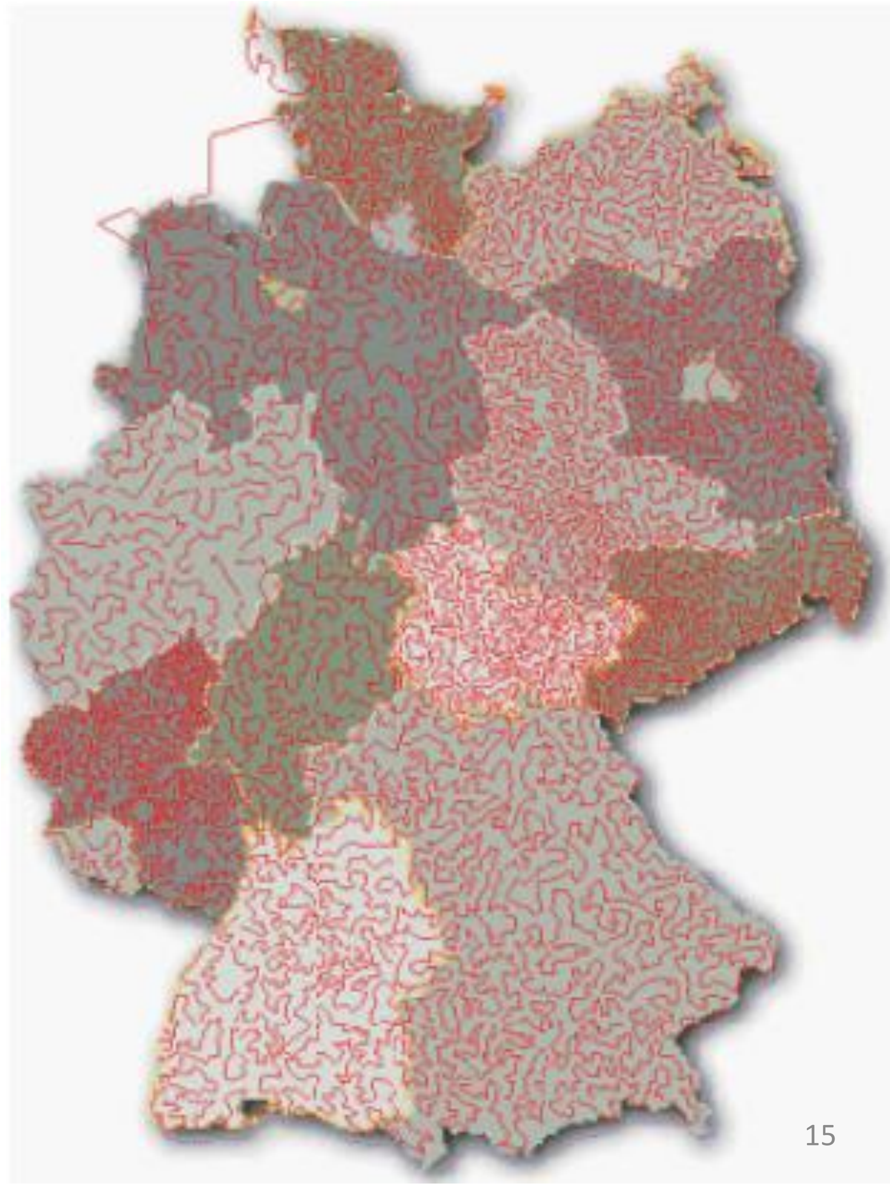
$$\frac{(n-1)!}{2}$$

El Problema del Vendedor Viajero

- ¿Cuánto uno tardaría en encontrar una solución para un problema de 50 ciudades?
  - Supongamos que se pueden chequear 1,000,000 de combinaciones por segundo.
  - Tenemos ca. 31449600 segundos en un año.
  - Pero para llegar a …….
    304140932017133780436126081660647688443776415689605120000

El Problema del Vendedor Viajero

- 15,112 ciudades en Alemania.
- Una red de 110 computadores de la universidades de Rice y Princeton.
- Tiempo total de cómputo de 22.6 años en un ordenador de 500 MHz
- Longitud: 66.000 km. (más que una vuelta a la tierra)



El Problema del Vendedor Viajero

- Hay extensiones:
  - En vez de un vendedor hay múltiples vendedores. Convencionalmente, de TSP pasa a llamarse m-TSP.
  - También en vez de vendedores pueden pensarse en camiones con capacidad. Este problema se llama el problema de ruteo de vehículos (VRP).
  - Es posible que las ciudades tengan también ventanas de tiempo donde pueden ser atendidas.
  - Costos simétricos y asimétricos.

El Problema del Vendedor Viajero

- El problema del TSP generalmente se ve como un grafo G=(V,E), donde V es un conjunto de m ciudades, $V=\{v_1,\ldots,v_m\}$. E es el conjunto de arcos $E = \{(r,s):r,s \in V\}$.
- E es normalmente una matriz asociada con una distancia (costo). Esta matriz D se define $D= (d_{r,s})$.
- En el caso simétrico, $d_{s,r} = d_{r,s}$.

El Problema del Vendedor Viajero

- El objetivo o función objetivo es encontrar una permutación P de las ciudades que minimice:

$$C(P) = \sum_{i=1}^{n-1} d_{P(i),P(i+1)} + d_{P(n),P(1)}$$

- Una permutación también se llama un tour.

El Problema del Vendedor Viajero

- Hay muchas heurísticas que intentan encontrar soluciones aproximadas:
  - El vecino más cercano
  - Greedy
  - 2-opt
  - 3-opt

El Problema del Vendedor Viajero

- El vecino más cercano consiste en siempre visitar el vecino que está más cerca.
- La complejidad es $O(n^2)$.
  1. Escoge una ciudad aleatoria.
  2. Encuentra la ciudad más cercana e ir ahí.
  3. ¿Quedan ciudades sin visitar? Sí, ir al paso 2.
  4. Retornar a la primera ciudad.

El Problema del Vendedor Viajero

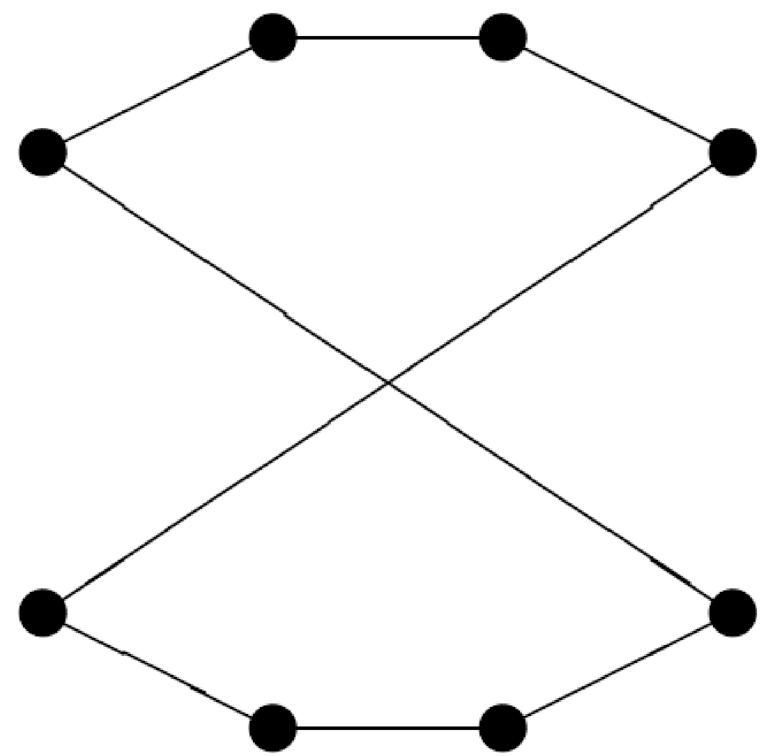
- Siempre que se parte de la misma ciudad, se obtiene la misma solución.
- Este algoritmo puede tener una variante: escoger la próxima ciudad de acuerdo a una probabilidad inversamente proporcional al costo.
- Normalmente tiene sus tours cerca del 25% de la cota inferior de Help-Karp.

El Problema del Vendedor Viajero

- Los algoritmos greedy siempren optan por la mejor solución inmediata (local), mientras buscan una solución.
  - Para un conjunto muy restricto de problemas encuentran la mejor solución global.
- El algoritmo greedy construye gradualmente un tour a través de la selección repetitiva del arco más corto agregándolo al tour siempre y cuando no se generen ciclos.
  1. Ordenar los arcos.
  2. Selecionar el arco más corto siempre que no genere ciclos.
  3. Tenemos N arcos en el tour? No, repite el paso 2.
- Un ciclo es un tour que no incluye todas las ciudades.
- 15-20% de la cota de Held-Karp.

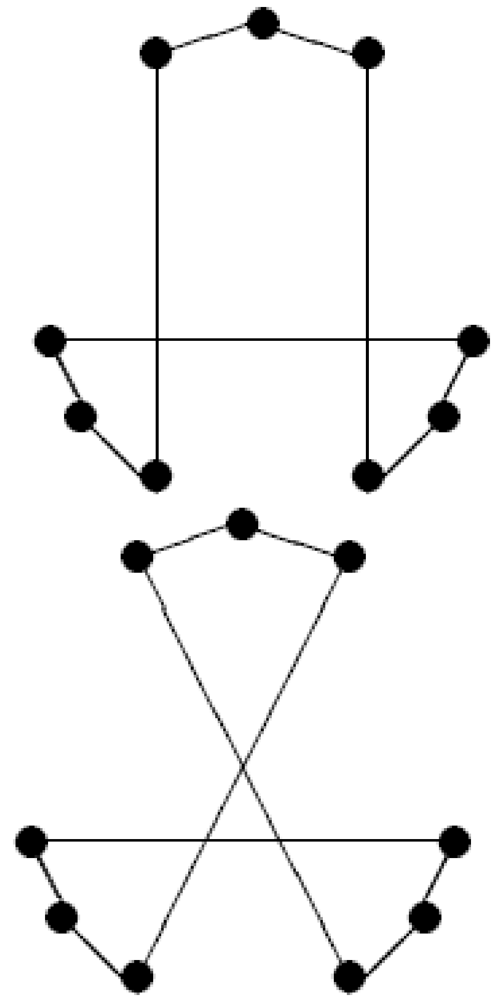
El Problema del Vendedor Viajero

- **2-opt:** La idea es cortar dos arcos e intercambiar los destinos.
- Hay una sola forma de re-conectarlos.
- Queda en su nueva forma si y sólo si mejora.
- Cuando no hay más mejora posible termina.
- Normalmente, 5% sobre la cota inferior de Held-Karp.



El Problema del Vendedor Viajero

- **3-opt**: se remueven tres arcos.
- Hay dos formas de reconectarlos.
- Frecuentemente, cerca del 3% de la cota de Help-Karp.



El Problema del Vendedor Viajero

- Una forma de mejorar la performance de 2-opt, que es $O(n^2)$:
  - Por cada arco que se escoge $d_{s,r}$ hay que revisar otro arco $d_{s',r'}$ completando el movimiento si y sólo si la suma de estos dos arcos es mayor a $d_{r,s'} + d_{s,r'}$
  - Se puede tener una lista de vecinos cercanos.
  - La función objetivo no es necesario recalcularla por completo.

El Problema del Vendedor Viajero

- La cota de Held-Karpp se obtiene mediante una relajación del problema.
- Cuando el problema se relaja, se obtiene una cota inferior al problema no relajado, es decir la solución del problema original no puede ser mejor a la del relajado.
- La cota se obtiene mediante la relajación lineal del problema de programación entera y el algoritmo Simplex y un algoritmo polinomial de separación de restricciones.

El Problema del Satisfactibilidad Booleana

- El problema de satisfactibilidad (SAT) es un problema de lógica matemática y la teoría de la computación.
- La satisfactibilidad proposicional es el problema de decidir si existe una asignación de zeros y unos a las variables que la hace verdadera.
- **Ejemplo**: La asignación de valores de verdad que satisfacen la fórmula: (P OR NOT(Q)) AND (Q OR R) AND (NOT(R) OR NOT(P)) es P=Q=1 y R = 0.

El Problema del Satisfactibilidad Booleana

- El juego de Sudoku también puede ser visto como un problema de satisfactibilidad con 729 variables proposicionales y una fórmula de 8829 claúsulas.

		1						
		2		3				4
			5			6		7
5			1	4				
	7						2	
				7	8			9
8		7			9			
4				6		3		
						5		

El Problema del Satisfactibilidad Booleana

@inproceedings{DBLP:conf/isaim/LynceO06,
author = {In{\^e}s Lynce and Jo{\"e}l Ouaknine},
title = {Sudoku as a SAT Problem},
booktitle = {ISAIM},
year = {2006},
ee = {http://anytime.cs.umass.edu/aimath06/proceedings/P34.pdf} }

Hay más Problemas Combinatorios

- El **problema de la mochila** (Knapsack Problem): Dado una mochila de capacidad C, escoger un subconjunto de objetos s $\subseteq$ S, tal que los elementos en s caben en la mochila, y además, maximiza el valor del contenido almacenado (cada objeto tiene un valor).
- El problema de **bin packing** consiste en minimizar el número de compartimientos necesarios para almacenar un número de objetos de diferentes volúmenes.
- El **coloreo de grafos** consiste en pintar los nodos de un grafo de manera de que dos nodos no tengan el mismo color, pero al mismo tiempo, utilizando el menor número de colores.

El Problema de la Mochila

No se puede mostrar la imagen en este momento.

El Problema de la Mochila

- Supongamos que N=|S|, es decir la cantidad de objetos disponibles para ser llevados.
- Una representación factible es un vector de N componentes binarias:
  - Por ejemplo: {0,1,0,1} para el caso de cuatro elementos.
- En este representación, cada componente indica si se lleva o no un elemento.
- La calidad de esta solución está determinada por el valor de las componentes indicadas con "1".
- La validez de la solución está dada por la capacidad total de los elementos marcados con "1".

El Problema de la Mochila

- Supongamos que tenemos una mochila con capacidad C=16.
- Además, que tenemos cuatro objetos con las siguientes características:

s	V_s	C_s	V_s/C_s
1	\$45	3	\$15
2	\$30	5	\$6
3	\$45	9	\$5
4	\$10	5	\$2

Fuerza Bruta

- El algoritmo de fuerza bruta prueba todas las combinaciones, es decir recorre todo el espacio de búsqueda, sin importar si está examinando zonas que son inválidas.
- En el ejemplo que la mejor solución deja espacio libre.
- Vemos que combinaciones inválidas también son exploradas.

Solución	Capacidad Usada	Valor Llevado	¿Válido?
{0,0,0,0}	0	0	Si
{0,0,0,1}	5	10	Si
{0,0,1,0}	9	45	Si
{0,0,1,1}	14	55	Si
{0,1,0,0}	5	30	Si
{0,1,0,1}	10	40	Si
{0,1,1,0}	14	75	Si
{0,1,1,1}	19	85	No
{1,0,0,0}	3	45	Si
{1,0,0,1}	8	55	Si
{1,0,1,0}	**12**	**90**	**Si**
{1,0,1,1}	17	100	No
{1,1,0,0}	8	75	Si
{1,1,0,1}	13	85	Si
{1,1,1,0}	17	120	No
{1,1,1,1}	22	130	No

Bactracking

- Es un algoritmo para encontrar todas (o algunas) de las mejores soluciones a un problema computacional.
- Construye incrementalmente soluciones y abandona una solución parcial tan pronto como determina que ésta no sirve para completar una solución válida.
  - Por ejemplo, en el problema de las n-reinas cualquier solución parcial formada por dos reinas que se atacan mutuamente es abandonada.

Bactracking

- Es generalmente más rápido que fuerza bruta ya que puede eliminar un gran número de candidatos que no son válidos.
- Para hacer backtracking es fundamental tener un orden de las variables y para cada variable tener un orden de instanciación.
- Conceptualmente, las soluciones parciales son nodos en un árbol.
  - Cada solución parcial es el padre de otro candidato. Ambos difieren en un solo paso.
  - El algoritmo va incrementalmente hasta que las soluciones parciales se transforman en soluciones completas.

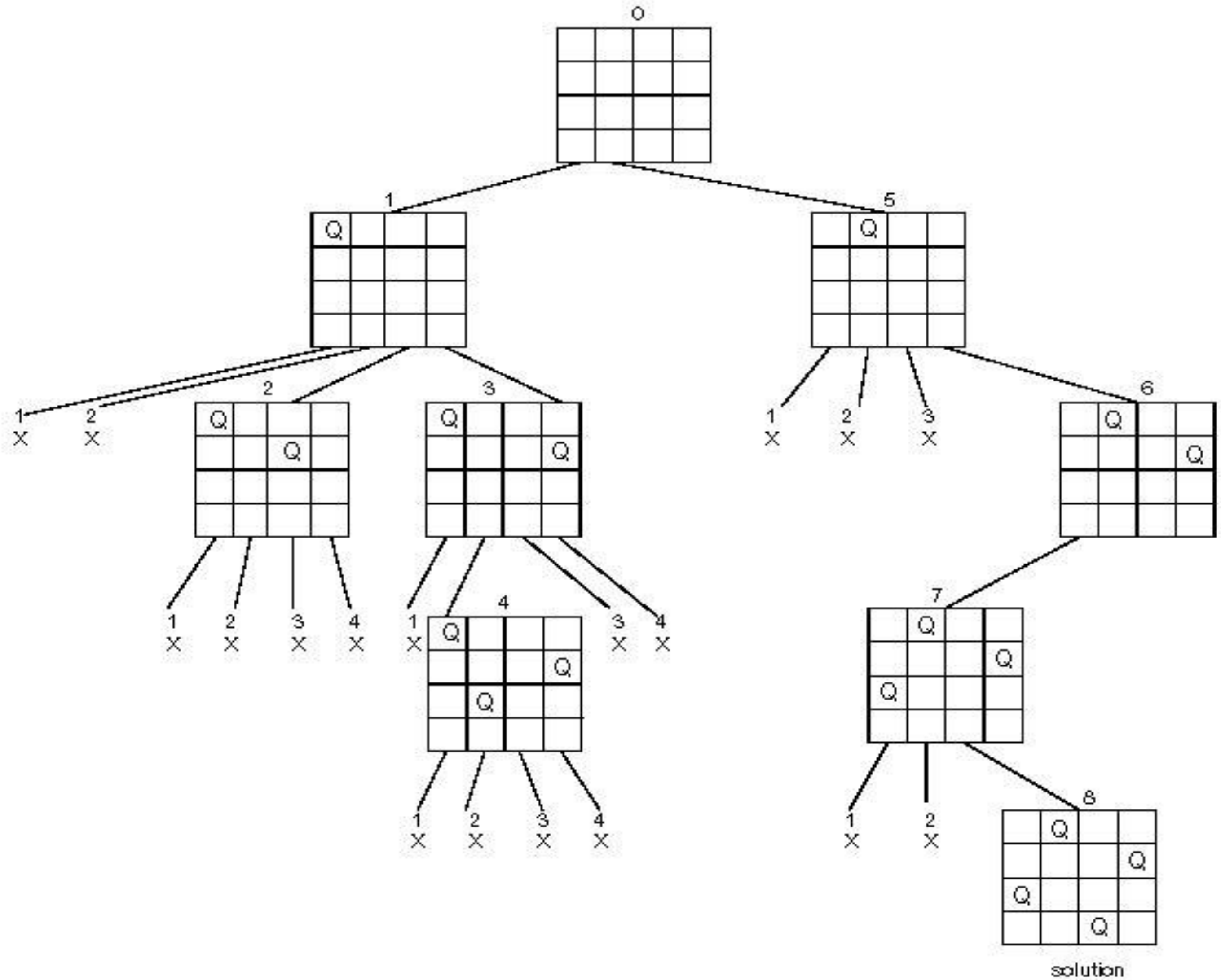
Bactracking

- Normalmente, backtracking recorre el árbol de manera recursiva, de la raíz hacia abajo (depth-first order).
- En cada nodo del árbol, el algoritmo chequea si la solución parcial puede completarse en una solución válida.
  - Sino puede, todos los sub-árboles son podados
  - Si lo es, sigue explorando los sub-árboles, de no ser una solución completa (hojas del árbol).

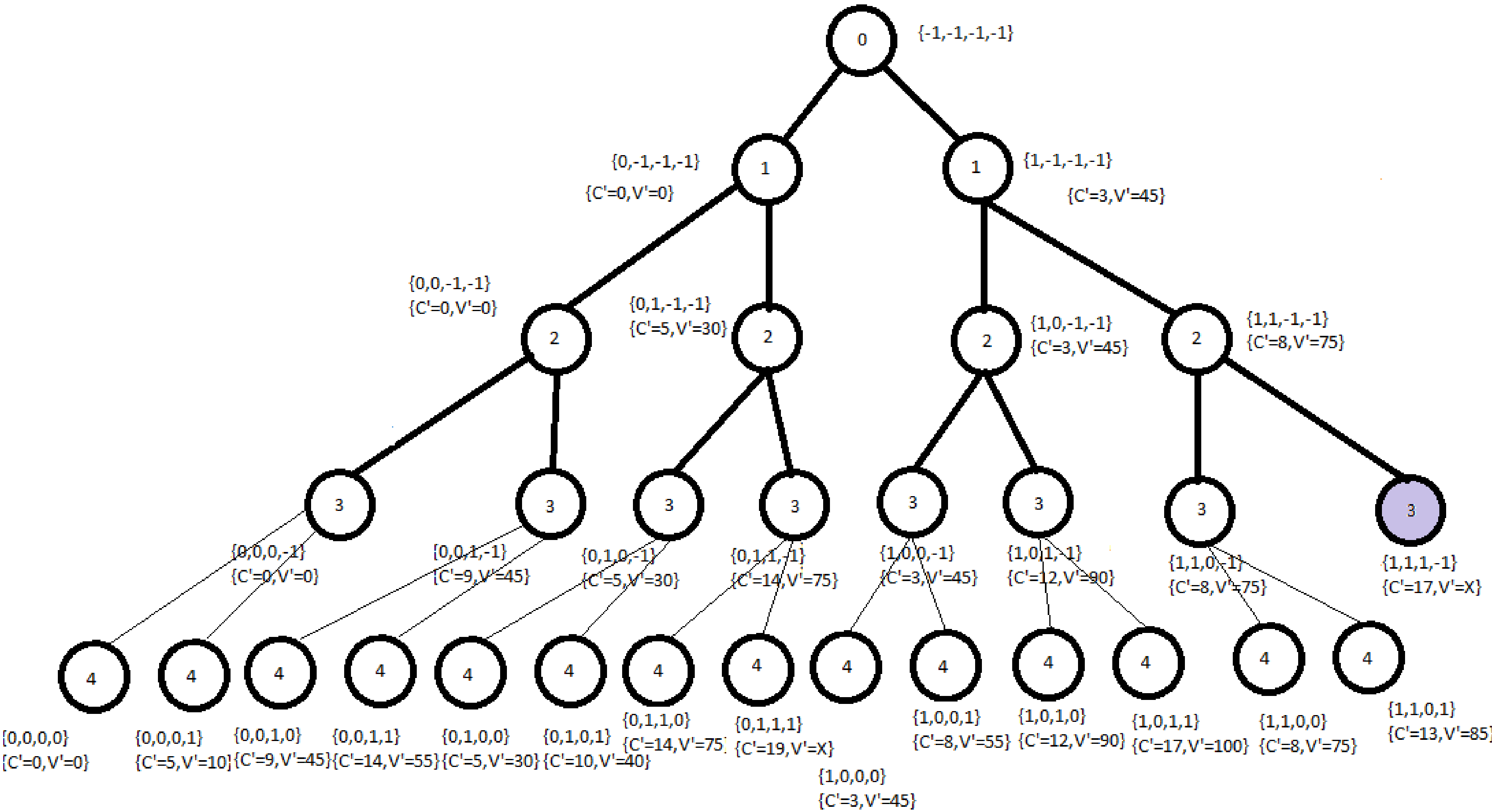
Bactracking

- En backtracking se explora cada nodo sistemáticamente:
  - backtracking(**nodo**):
    1. Retorna falso si la solución parcial en **nodo** no es válida o no vale la pena completarla.
    2. Muestra la solución en **nodo** si es válida.
    3. Escoge el s como el primer valor posible de los que se pueden obtener.
    4. Mientras existe un s hacer
        a. Backtracking(s)
        b. S se instancia con el próximo valor para el nodo.

Backtracking



Bactracking



Branch and Bound

- Es una mejora a backtracking útil para problemas de optimización.
- La idea es utilizar una función de acotamiento que permite calcular las cotas de los sub-árboles.
  - De esta forma determinamos cuán prometedoras son las soluciones parciales.
- Si dice que es prometedor se expande el árbol, sino se poda.
- Si es prometedor o no depende del valor de la mejor solución encontrada hasta ahora.

Branch and Bound

- Una de las dificultades para Branch and Bound es encontrar buenas funciones de acotamiento.
- Por ejemplo, para el caso del problema de la mochila, podemos pensar en una solución greedy.
  - Esta solución encuentra el óptimo si los objetos pudieren fraccionarse.
  - El algoritmo greedy agrega objetos a la mochila hasta que el próximo no quepa por completo. De este último se agrega sólo la fracción que cabe.

Branch and Bound

- Entonces para el problema de la mochila, la cota está dada por:
  - Bound = valorActual + valor de los objetos que pueden ser totalmente agregados + (C – tamañoTotal) * densidadDelElementoFraccionado.
  - El tamañoTotal o capacidad total utilizada está dado por la suma de los volúmenes de los elementos ya agregados + el volumen de los que se pueden agregar
- Al decir que es “greedy” indicamos que vamos agregando en orden decreciente a su valor.

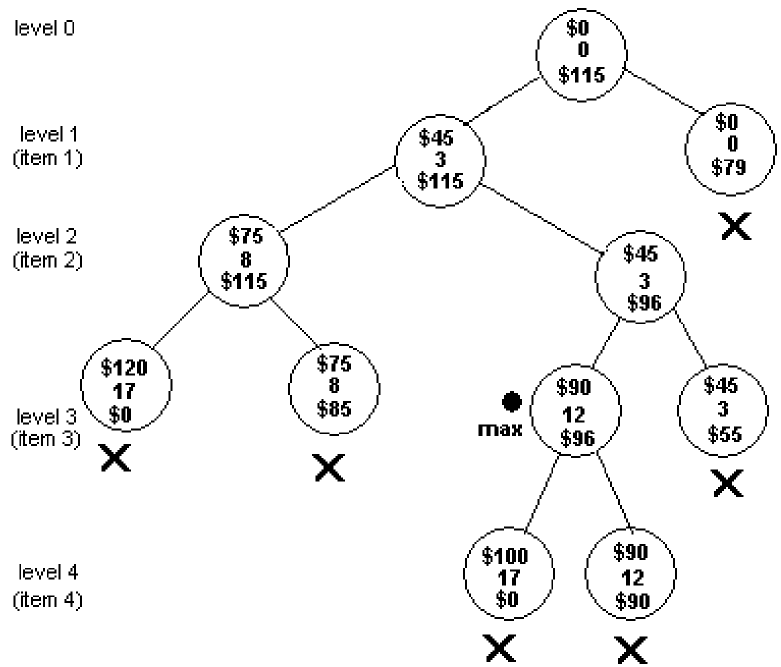
Branch and Bound

- Para nuestro ejemplo, el algoritmo greedy va:
  - Primero pone el primer objeto, ocupando 3/16 de la capacidad (valor $45),
  - Después, pone el segundo objeto, ocupando 8/16 de la capacidad (valor $75).
  - Después, pone el tercer objeto, ocupando las ocho unidades que le quedan de capacidad. Dado que esto es sólo 8/9 del objeto, el valor máximo alcanzable es $115.
  - Ninguna solución puede exceder ese valor, pero esta solución no es válida.

Branch and Bound

- En vez de hacer una búsqueda en profundidad como backtracking, Branch and Bound lo hace a lo ancho.
- Esto nos permite tener las cotas para todos los hijos de un nodo, y saber de inmediato cuales podar y cuales explorar.

Branch and Bound

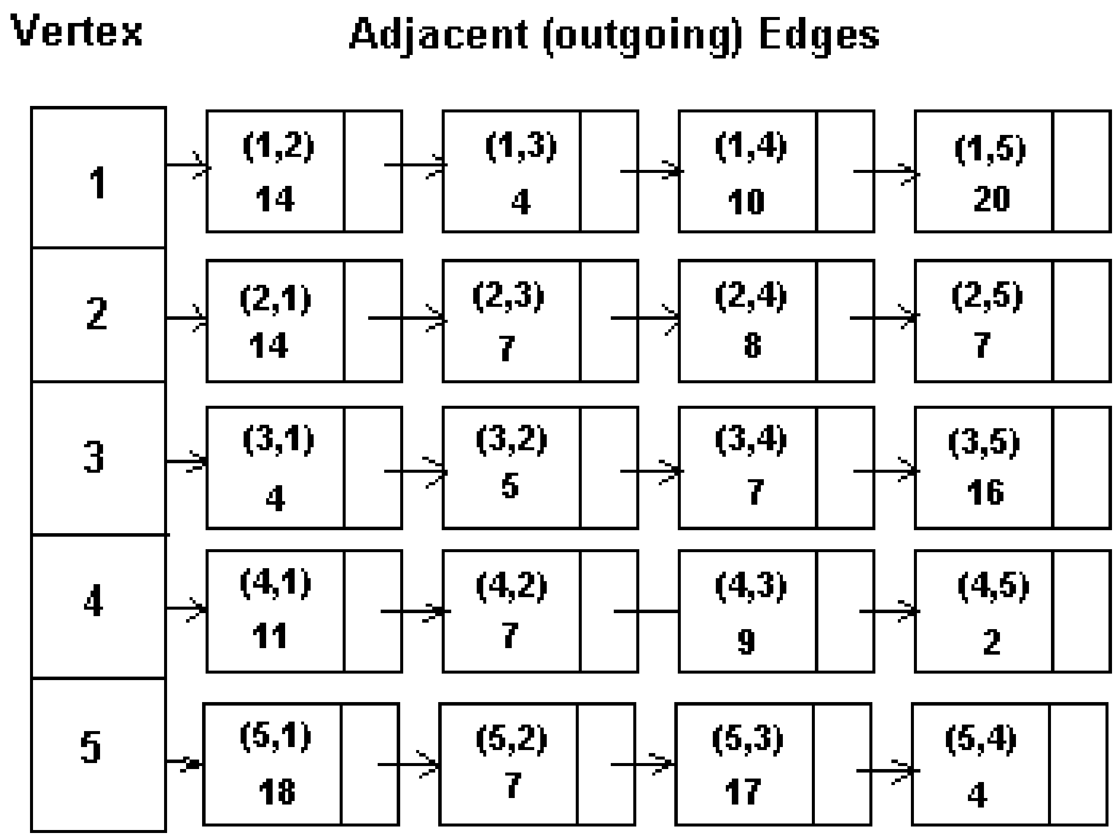


Source:
http://www.academic.marist.edu/~jzbv/algorithms/Branch%20and%20Bound.htm

Branch and Bound

- La función de acotamiento para el problema del vendedor viajero está dada por la siguiente observación:
  - "Al menos tenemos que salir de cada ciudad una vez. Por ende, podemos encontrar una cota para la solución:
    - Relajando la restricción de que debemos visitar todas las ciudades.
    - Asumiendo que siempre salimos del nodo utilizando el arco de menor coste de las ciudades que restan por visitar."

Branch and Bound



Source:
http://www.academic.marist.edu/~jzbv/algorithms/Branch%20and%20Bound.htm

Branch and Bound

- Podemos asumir que partimos de cualquier nodo, por ejemplo del “1”.
- Entonces, los vértices de menor coste para cada nodo van a estar dados por:
  - Para el vértice 1: min(14,4,10,20)=4
  - Para el vértice 2: min(14,7,8,7)=7
  - Para el vértice 3: min(4,5,7,16)=4
  - Para el vértice 4: min(11,7,9,2)=2
  - Para el vértice 5: min(18,7,17,4)=4
- Por ende, la cota es 4+7+4+2+4=21. No es posible armar un tour con un coste menor que ese ya que hemos relajado restricciones.

Branch and Bound

- Calculemos la primera expansión del árbol a lo ancho:

Tour parcial	Coste fijo	Estimación Resto	Cota
1→2	14	7+4+2+4	31
1→3	**4**	**7+4+2+4**	**21**
1→4	10	7+4+2+4	27
1→5	20	7+4+2+4	37

Las cotas muestran que es mejor explorar los tours que comienzan con 1,3. Si todos los tour bajo esa rama del árbol de exploración tienen un coste menor o igual a 27, no merece la pena explorar los tours que comienzan con 1,2 ó 1,4 o 1,5.

Branch and Bound

- Calculemos la segunda expansión del árbol a lo ancho:

Tour parcial	Coste fijo	Estimación Resto	Cota
1→3→2	**4+5**	**7+4+2**	**22**
1→3→4	**4+7**	**7+4+2**	**24**
1→3→5	4+16	7+4+2	33

Hay que actualizar las cotas ya que no se puede volver al nodo 3:

Para el vértice 2: min(14,8,7)=7

Para el vértice 4: min(11,7,2)=2

Para el vértice 5: min(18,7,4)=4

Encontramos dos caminos que son menores al “27” del 1,4. Por ende, hay que seguir explorando esta rama.

Branch and Bound

Tour Completo	Coto Total
1→3→2->4->5->1	37
1→3→2->5->4->1	31

Al fijar la cuarta ciudad, sabemos que debemos visitar la restante y de ahí regresar, por ende, obtenemos el valor del tour completo. Por ende, no necesitamos hacer estimaciones de las cotas.

En ambos casos, los valores son mayores ala cota obtenida por la solución parcial 1,3,4 (24). Ergo, debemos explorar esta rama:

Tour Completo	Coto Total
1→3→4->2->5->1	43
1→3→4->5->2->1	34

Branch and Bound

- Dado que todos los tours completos siguen siendo de coste mayor al estimado por la solución parcial 1,4 (27), debemos expandir esa rama del árbol:

Tour parcial	Coste fijo	Estimación Resto	Cota
1→4→2	10+7	7+4+7	32
1→4→3	10+9	7+4+7	34
1→4→5	**10+2**	**7+4+7**	**27**

Hay que actualizar las cotas ya que no se puede volver al nodo 4:

Para el vértice 2: min(14,7,7)=7

Para el vértice 3: min(4,5,16)=4

Para el vértice 5: min(18,7,17)=7

Encontramos que el tour parcial 1,4,5 es la mejor solución parcial hasta ahora, es mejor que todos los tour completos y además conservo el valor de la cota. Por ende, hay que seguir explorando esta rama.

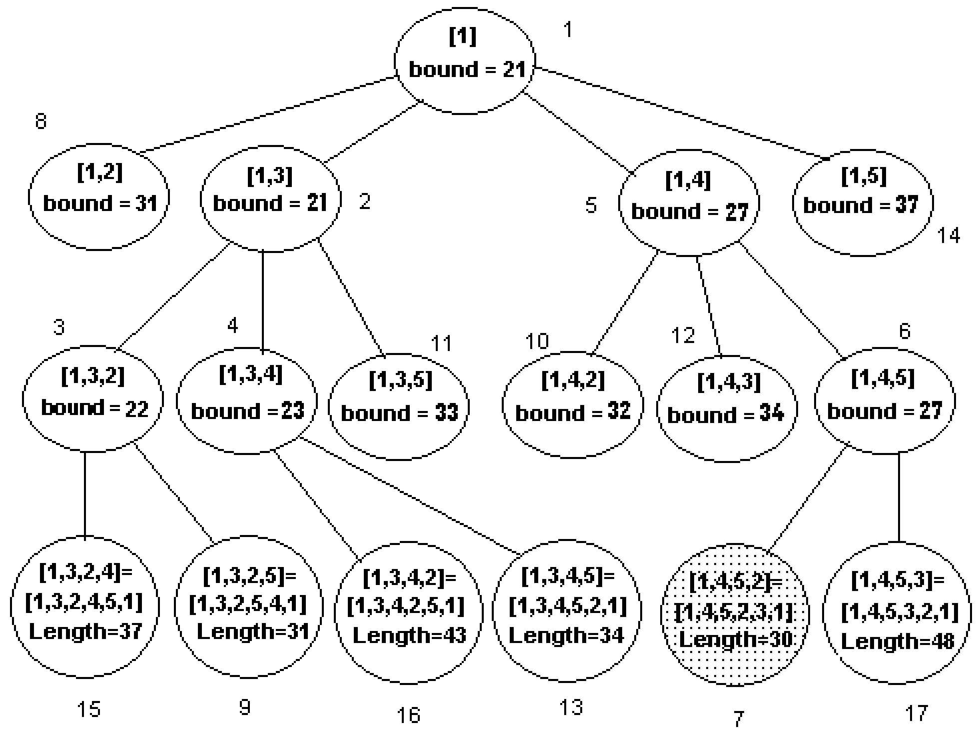
Branch and Bound

- Al agregar el siguiente nodo, nuevamente, no es necesario calcular las cotas ya que la construcción del tour es automática.

Tour Completo	Coto Total
1→4→5->2->3->1	**30**
1→4→5->3->2->1	48

No tenemos mejor solución parcial, ni completa, por ende, hemos encontrado la solución óptima del problema. Es decir, no es necesario seguir explorando.

Branch and Bound



Construcción vs. Mejora de Soluciones

- Por un lado, vemos que técnicas como Branch and Bound y Backtracking van construyendo soluciones factibles mientras recorren el espacio de búsqueda.
  - Fuerza Bruta construye todas las soluciones tanto factibles como no-factibles.
- Por otro lado, vemos que técnicas como 2-opt y 3-opt parten de una solución, en lo posible factible, y van haciendo cambios a esta solución con el objetivo de mejorarla manteniendo, en lo posible, la factibilidad.

Hill Climbing

- Es una técnica de búsqueda local.
- Es un algoritmo iterativo que comienza con una solución arbitraria e intenta mejorarla de manera incremental.
- Las mejoras son mediante cambios locales.
- Si los cambios producen una mejor solución, entonces ésta reemplaza la inicial.
- Se repite hasta que no hayan más cambios o un número de iteraciones definido se alcance.

Hill Climbing

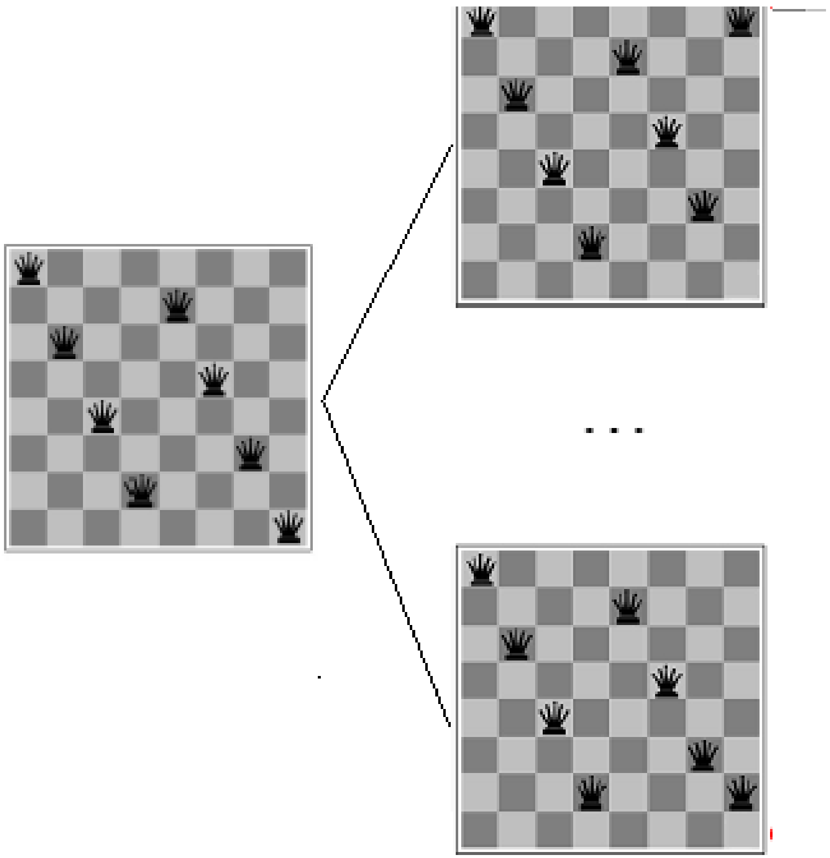
- Por ejemplo, si lo aplicamos al problema del vendedor viajero, basta con intercambiar dos ciudades en el tour para generar una nueva solución.
- Por ejemplo:
  - {0,3,2,5,4,1,0} => {0,4,2,5,3,1,0}
- ¿Cómo se escogen los pares?
  - Los arcos más caros, aleatorio
  - Se puede utilizar 2-opt ó 3-opt.

Hill Climbing

- El problema es que esta técnica cae en óptimos locales.
- Por ende, se reinicia muchas veces (Multi-start Hill Climbing).
- En resumen,
  1. Comenzar con alguna solución s
  2. Moverse al vecino t que tenga mejor solución
  3. s=t
  4. Ir a 2 hasta cierto número de iteraciones o no haya mejor vecino t.

Hill Climbing

- En el caso del problema de las ocho reinas, podemos escoger la que genere más conflictos.

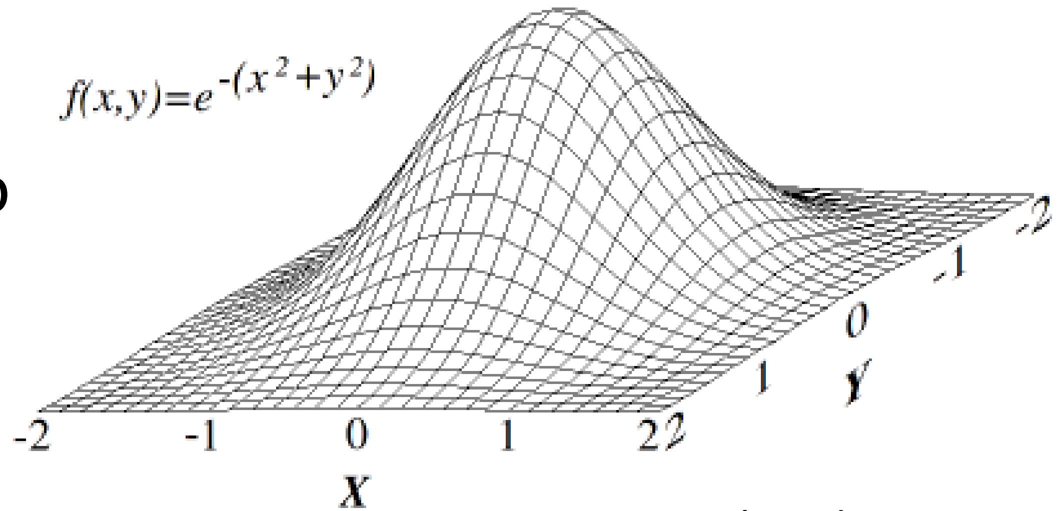


Hill Climbing

- ¿Cómo se escoge el vecino?
- El mejor es "greedy", pero ¿Siempre es necesario escoger el mejor?
- ¿Qué pasa sino hay mejor vecino? ¿Parar?
- **Stochastic Hill Climbing** no examina todos los vecinos antes de decidir cómo moverse. Más bien, selecciona un vecino aleatoriamente. Cada vecino tiene una probabilidad dependiendo de su mejora.

Hill Climbing

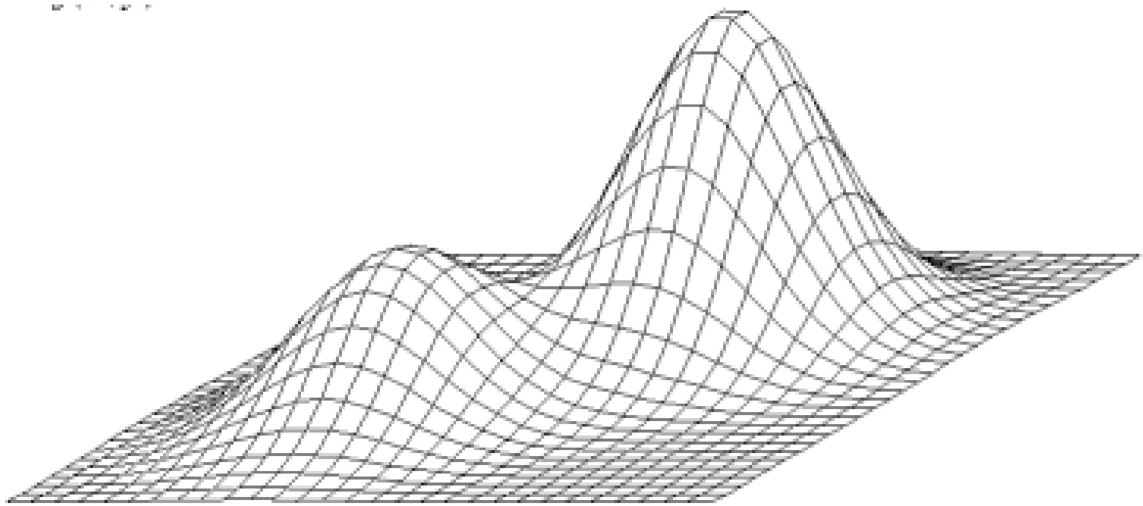
- Queda estancado en un óptimo local al menos que el espacio de búsqueda sea cóncavo
- Los planicies también son un problema para este algoritmo, ya que todos los vecinos se ven iguales.



Fuente: Wikipedia

Hill Climbing

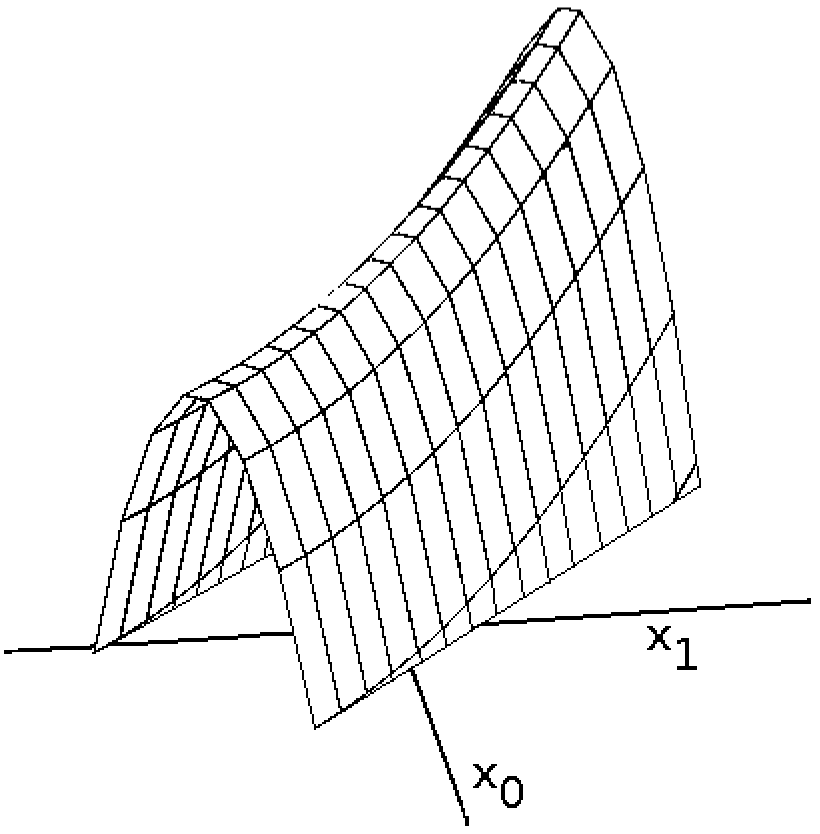
- Prácticamente todo el tema con las heurísticas de búsqueda es lidiar con los óptimos locales.
- Debido al tamaño, no se puede ver todo el espacio de búsqueda de una vez, sino sólo una parte.



Fuente: Wikipedia

Hill Climbing

- Los caballetes también son un problema porque como se modifica sólo una parte de la solución, se puede estar moviendo de un lado al otro, haciendo la convergencia más lenta.
- Hay pocas chances de hacer movimientos que mejoren rápidamene.



Fuente: Wikipedia

Hill Climbing

- Nótese que a veces es necesario empeorar las soluciones un poco para poder mejorarlas mucho más después.
- Por ejemplo WALKSAT:
  1. Escoger una clausula no satisfecha de manera aleatoria.
  2. Considere 3 vecinos: cambiar cada variable
  3. Si alguna mejora, acepta la mejor, sino,
     - 50% de las veces seleccionar el menos malo.
     - 50% de las veces escoger uno aleatorio.
  4. Repetir hasta que algún criterio se cumpla.

Simulated Annealing

- Propuesto por Kirpatrick et al. en 1983.
- Permite empeorar la función objetivo para escapar óptimos locales.
- Es fácil hacer que funcione, pero difícil que funcione bien, ya que es necesario encontrar buenos parámetros.
- Es una técnica de búsqueda local.

Simulated Annealing

- Para evitar el estancamiento en un óptimo local, se permiten movimientos a soluciones peores.
- Estos movimientos de escape deben controlarse adecuadamente para no desviar la búsqueda cuando se dirija a una buena solución.
- Para ésto, se utiliza una función de probabilidad que disminuye a medida que la búsqueda avanza.

Simulated Annealing

- La probabilidad de un incremento energético δE a temperatura T es:
  - $P|\delta E| = e^{\delta E/kT}$
- Donde k es la constante de Boltzmann
- Dada una perturbación
  - Si disminuye la energía, se acepta.
  - Si aumenta la energía, se acepta con una probabilidad dada por la fórmula anterior.
- Los estados del sistema son soluciones factibles. Mientras la energía es el coste de la solución.
- Un cambio de estado es un movimiento factible.

Simulated Annealing

- Si f(x) es la función de costo y N(x) es el vecindario dado por algún movimiento pre-definido, el algoritmo se inicializa:
  - Generar una solución inicial x_0.
  - Escoger una temperatura inicial $t_0 > 0$.
  - Función de reducción de la temperatura α.
  - Escoger el número de repeticiones nrep.
  - Seleccionar el criterio de parada C.

Simulated Annealing

- **Repetir**
  - **Repetir**
    - Escoger aleatoriamente una solución x de $N(x_0)$.
    - $\delta := f(x) - f(x_0)$
    - **Si** $\delta < 0$ **Entonces** {x es mejor que x_0}
      - $x_0 := x$.
    - **Si no** {x es peor que x_0}
      - Generar aleatoriamente u de U(0,1).
      - **Si** $u < \exp(-\delta/t)$ **Entonces**
        - » $x_0 := x$.
      - **Fin Si**
    - **Fin Si**
  - **Hasta que** i=nrep
  - $t = \alpha(t)$
- **Hasta que** C se verifique

Simulated Annealing

- Normalmente, k no suele considerarse ya que no tiene equivalencia.
- Por otro lado, t es un parámetro que controla la probabilidad de movimientos de escape.
- Cuando t es pequeña, no habrán movimientos de escapes y la búsqueda acabará, frecuentemente, en un óptimo local.
- Resultado teórico:
  – Está comprobada la convergencia al óptimo cuando la temperatura se reduce lentamente.
  – Los tiempos de cómputo no son asumibles.

Simulated Annealing

- La temperatura inicial t_0 debe ser:
  - Independiente de la solución inicial.
  - Lo suficientemente alta como para aceptar casi libremente las soluciones de entorno.
- La temperatura final t_f debería ser 0, pero en la práctica el proceso converge antes:
  - Si es muy baja, se desaprovecha tiempo.
  - Si es muy alta, no logramos un óptimo local.

Simulated Annealing

- Existen varios mecanismos de enfriamiento α:
  - Descenso constante de temperatura.
  - **Geométrico**: $t_{i+1} = \alpha\ t_t\ \alpha \in [0.8, 0.99]$
  - **Boltzmann**: $t_i = t_o / (1 + \log(i))$
  - **Cauchy**: $t_i = t_o / (1 + i)$
  - **Ludi y Mees**: $t_{i+1} = t_i / (1+ \beta\ t)$, con β muy pequeño.
- La solución inicial puede ser obtenida de cualquier forma inclusive aleatoriamente.

Múltiples Objetivos

- Hasta ahora hemos pensado en que sólo hay un factor que optimizar. Pero, en la vida real, no es así y generalmente hay múltiples objetivos.
- El problema es que es difícil encontrar un óptimo para cada una de las k funciones de manera simultánea..
- Mejorar uno significa empeorar otros. Por ejemplo, velocidad y combustible en un avión.

Múltiples Objetivos

- Hay varias formas de solucionar este problema:
  - Eficiencia de Pareto.
  - Suma ponderada, es decir transformarlo a un solo objetivo donde se hace una suma ponderada de los diferentes objetivos.
  - Asignar prioridades donde se cuenta su importancia relativa durante el proceso de optimización.

Eficiencia de Pareto

- Es un conjunto de soluciones.
- Una solución es pareto-óptima cuando no existe otra solución que mejora un objetivo sin empeorar otro.

Referencias

- http://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas#Soluciones_al_problema_de_las_ocho_reinas
- http://es.wikipedia.org/wiki/Optimizaci%C3%B3n_multiobjetivo
- http://es.wikipedia.org/wiki/Eficiencia_de_Pareto
- http://en.wikipedia.org/wiki/Backtracking
- http://xlinux.nist.gov/dads//HTML/greedyalgo.html