Ricerca euristica

Review: Tree Search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

A strategy is defined by picking the order of node expansion

Best-first search

- **Idea: use an evaluation function for each node**
  - **estimate of "desirability"**
  - **→ Expand most desirable unexpanded node**

- **Implementation:**
  Order the nodes in fringe in decreasing order of desirability

- **Special cases:**
  - **greedy best-first search**
  - **A* search**

Funzioni euristiche

Dato un albero di ricerca, la funzione euristica h' associa un numero a ciascun nodo n, come segue:

$h'(n)$ = stima del costo del percorso ottimale dal nodo n ad un nodo goal

Si assume che h' sia non-negativa, e $h'(n)=0$ se e solo se n è un nodo goal.

Euristiche

Euristica (dal Greco "trovare", "scoprire"):

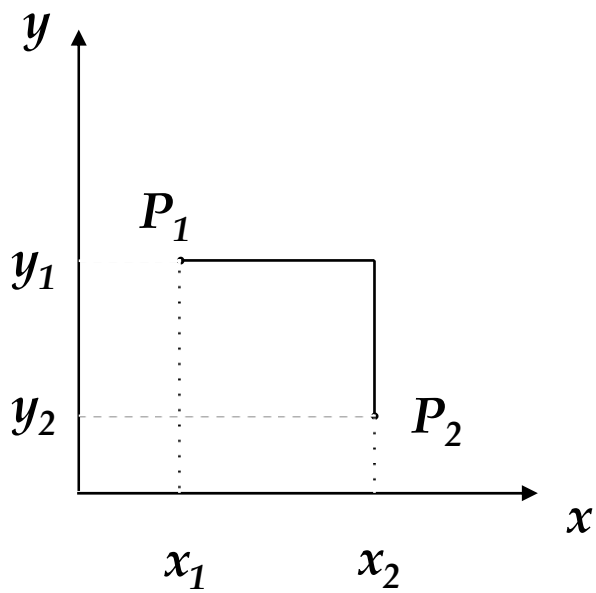
"è un criterio, metodo, o principio per decidere quale tra diverse azioni alternative promette essere la più efficiente al fine di raggiungere un certo obiettivo"

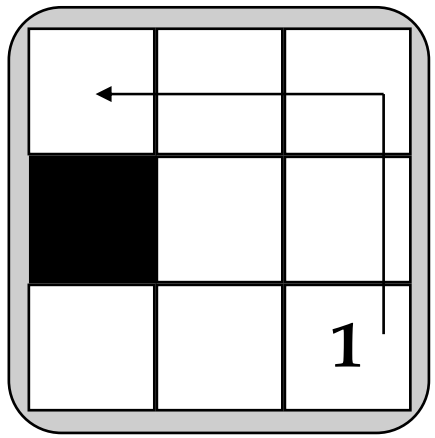
da *Heuristics* di Judea Pearl (1984).

Distanza di Manhattan

Dati due punti nel piano di coordinate $P_1=(x_1,y_1)$ e $P_2=(x_2,y_2)$, la *distanza di Manhattan* è definita come

$$D(P_1,P_2)= |x_2-x_1|+ |y_2-y_1|$$





Euristiche per il gioco dell'8

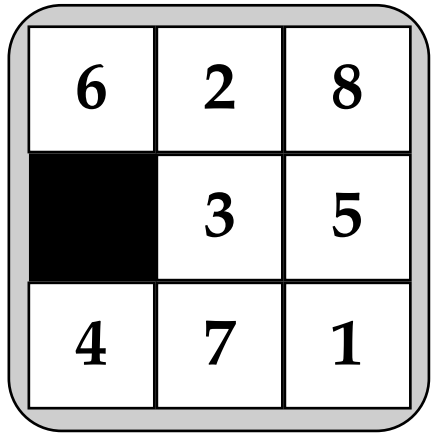
- $h'_1(n)$ = numero di tessere fuori posto

- $h'_2(n)$ = somma delle distanze di Manhattan per ciascuna tessera

Esempio:

$h'_1(n) = 7$

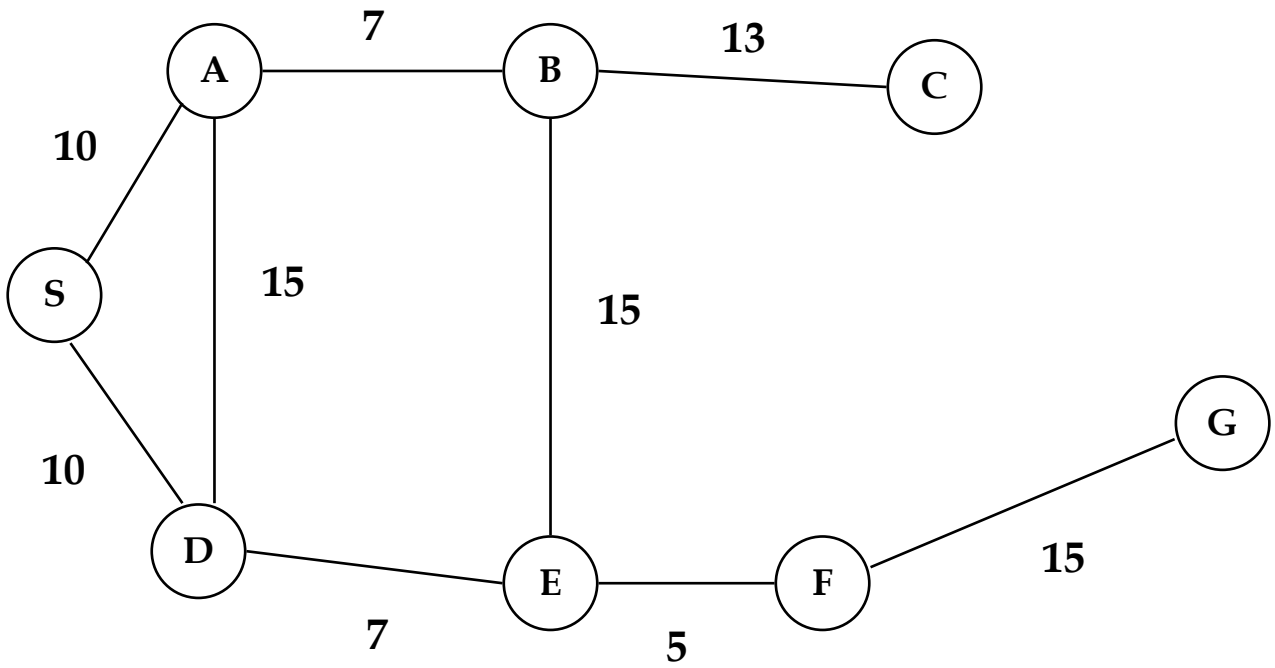
$h'_2(n) = 4+0+2+3+1+3+1+3 = 17$



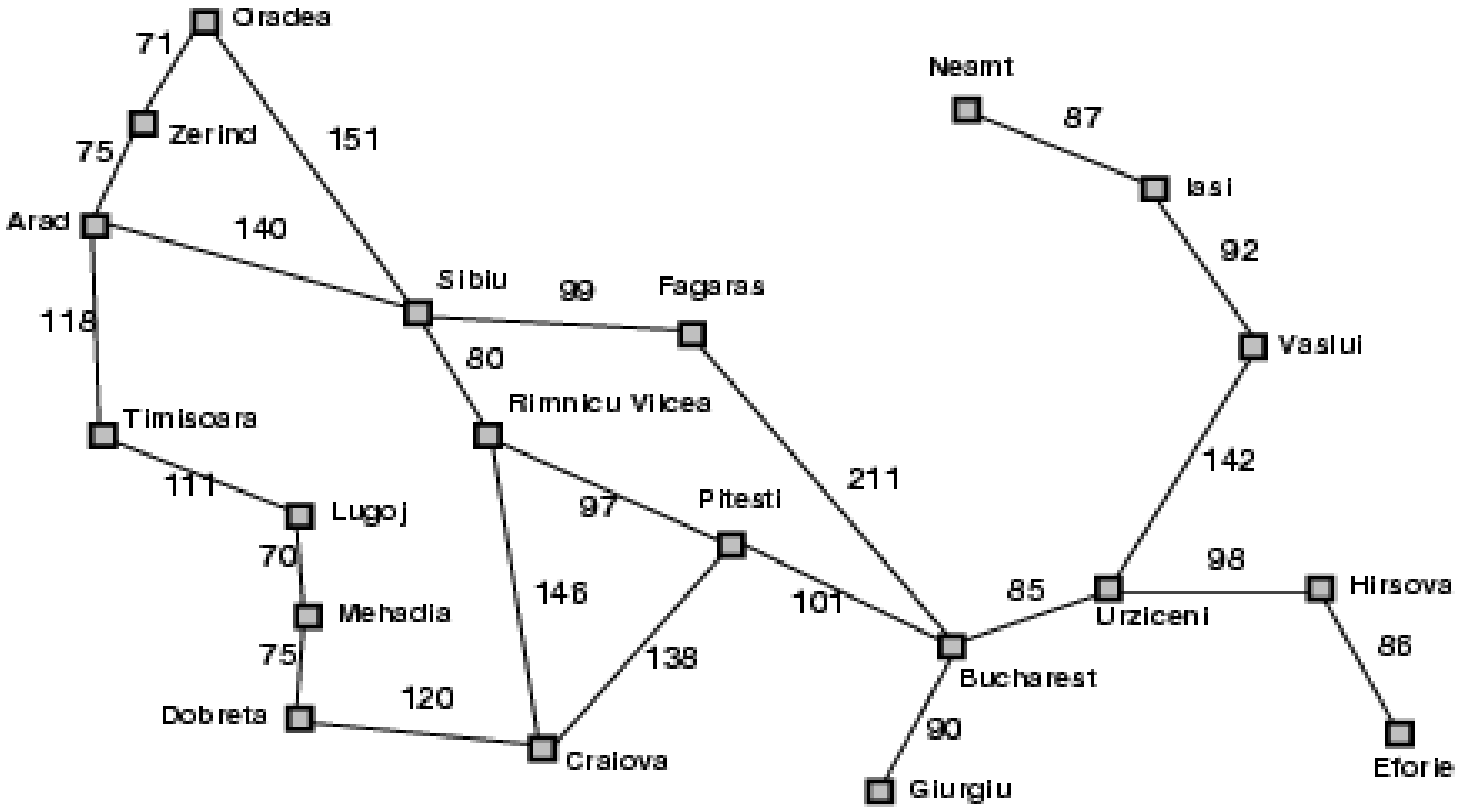
Altro esempio

Trovare un percorso da S a G. Una possibile funzione euristica è:

***h'(n)* = distanza in linea retta tra *n* e G**



Romania with step costs in km



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

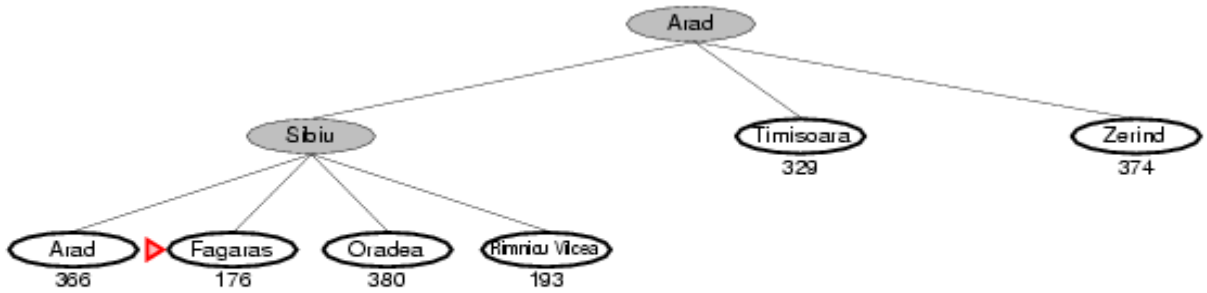
Ricerca greedy best-first

1) Sia *L* la lista dei nodi "iniziali"
2) Sia *n* il nodo di *L* per cui *h'(n)* è minima. Se *L* è vuota, ESCI
3) Se *n* è un nodo "goal", allora ESCI e restituisci in output *n* ed il percorso dal nodo iniziale fino a *n*
4) Altrimenti, cancella *n* da *L* ed aggiungi a *L* tutti i nodi discendenti di *n*, etichettando ciascuno di essi con il percorso dal nodo iniziale
5) Ritorna al passo 2)

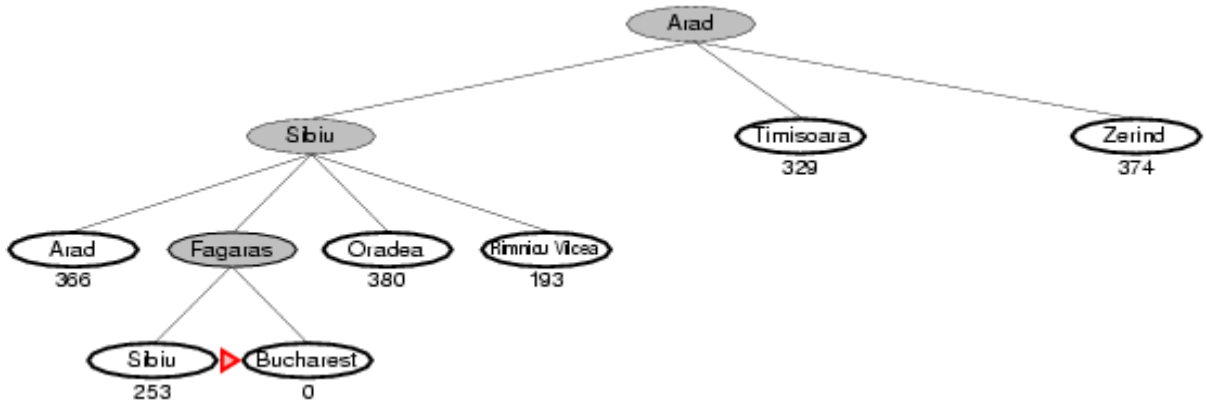
Greedy best-first search example

Greedy best-first search example

Greedy best-first search example



Greedy best-first search example



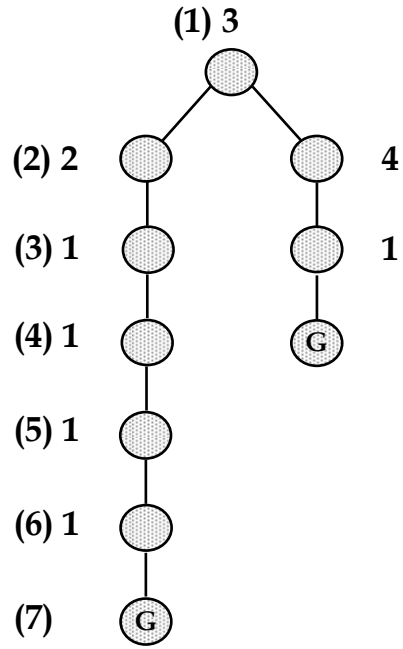
Ricerca greedy best-first
Completezza, ottimalità e complessità

La ricerca best-first è *non ottimale* e *incompleta*.

La complessità spaziale e temporale è *esponenziale* (una buona funzione euristica può ridurla drasticamente)

Esempio (non ottimalità):

I numeri in parentesi rappresentano l'ordine di espansione; gli altri il valore della funzione euristica *h'*



Properties of greedy best-first search

- **Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →**
- **Time? $O(b^m)$, but a good heuristic can give dramatic improvement**
- **Space? $O(b^m)$ -- keeps all nodes in memory**
- **Optimal? No**

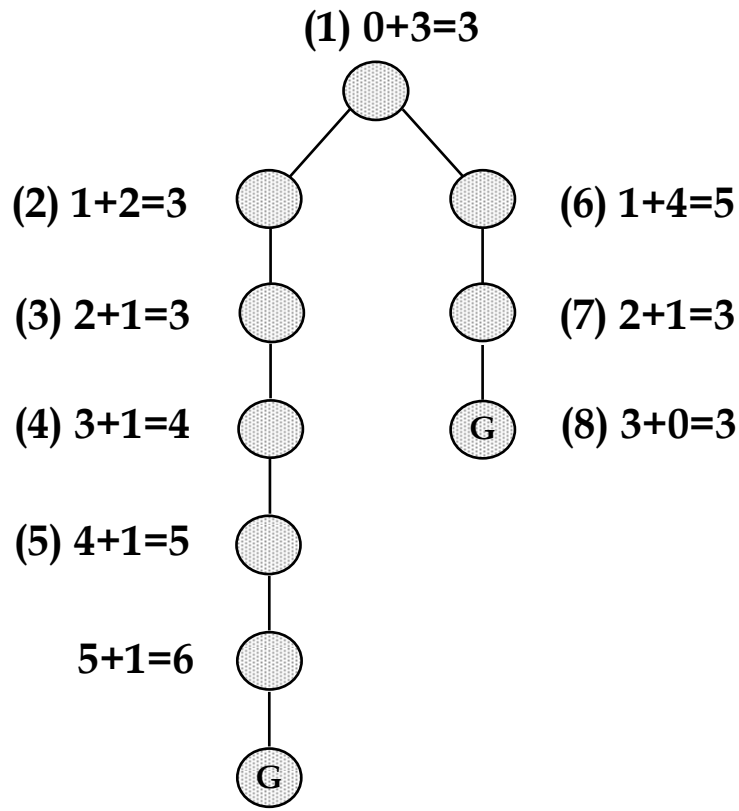
Algoritmo A*

L'algoritmo A* realizza una ricerca best-first con funzione di valutazione data da:

$f(n)=g(n)+h'(n)$

dove

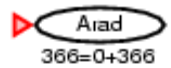
$g(n)$ = **costo per raggiungere il nodo n a partire dal nodo iniziale**



Algoritmo A*

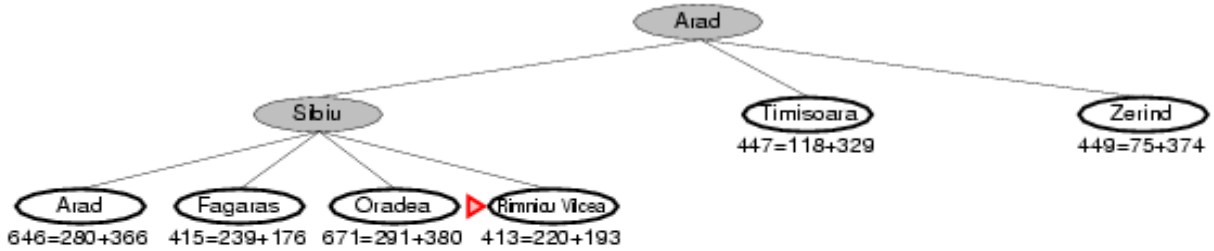
1) Sia *L* la lista dei nodi "iniziali"
2) Sia *n* il nodo di *L* per cui $f(n)=g(n)+h'(n)$ è minima. Se *L* è vuota, ESCI
3) Se *n* è un nodo "goal", allora ESCI e restituisci in output *n* ed il percorso dal nodo iniziale fino a *n*
4) Altrimenti, cancella *n* da *L* ed aggiung a *L* tutti i nodi discendenti di *n*, etichettando ciascuno di essi con il percorso dal nodo iniziale
5) Ritorna al passo 2)

A* search example

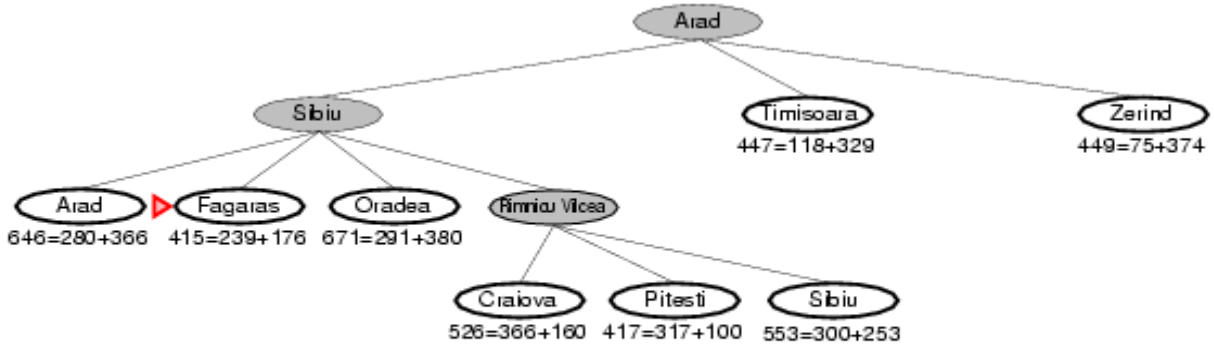


A* search example

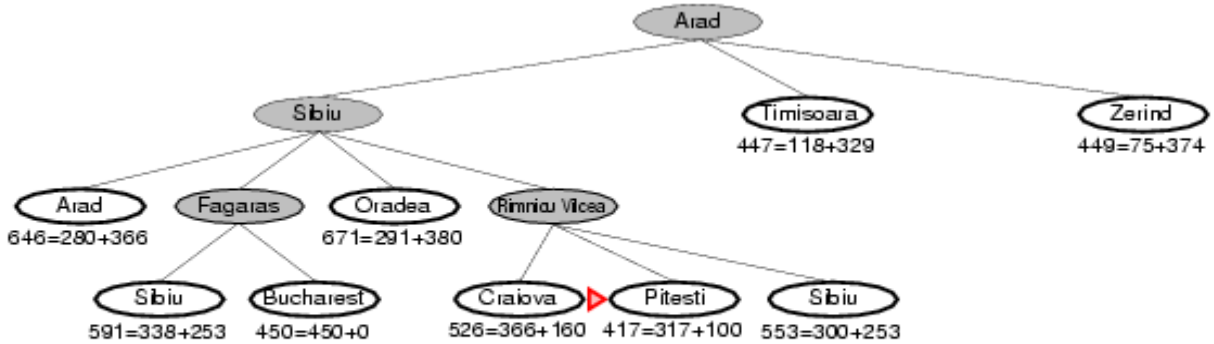
A* search example



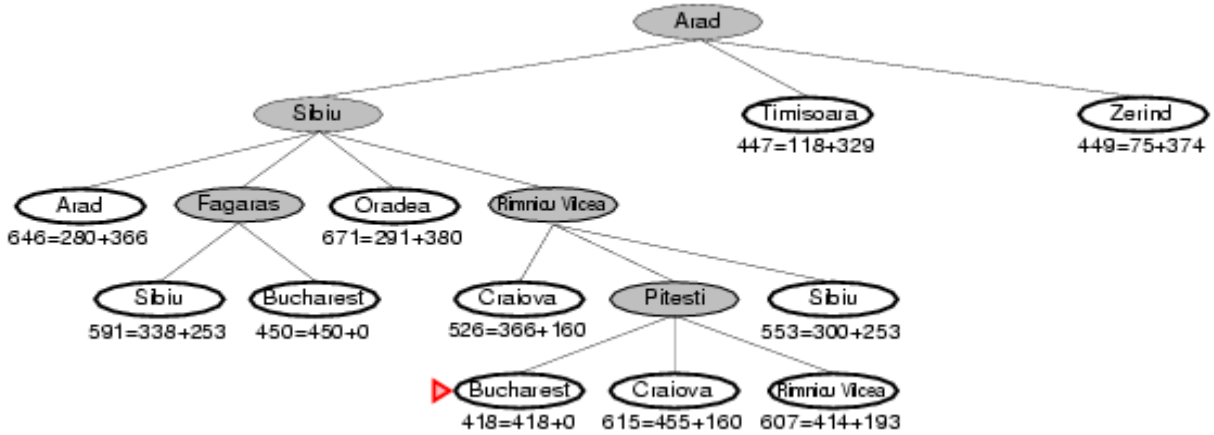
A* search example



A* search example

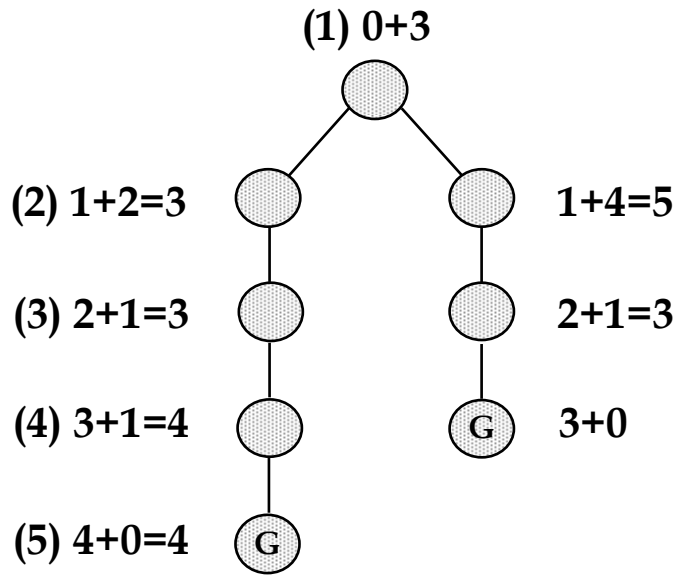


A* search example



Ottimalità di A*

In generale A* non è ottimale, a meno che la funzione euristica non verifichi una condizione particolare.



Ottimalità di A*

Teorema: **Se la funzione h' è *ammissibile*, cioè $h'(n) \leq h(n)$ per ogni n, allora l'algoritmo A* è ottimale.**

Dim: **Siano s un nodo ottimale e x uno non ottimale. Sia $n_0\ n_1\ ...\ n_k=s$ il percorso dalla radice a s. Per ogni $i=0...k$, si ha $f(n_i) < f(x)$. Infatti:**

$$f(n_i) = g(n_i) + h'(n_i) \leq g(n_i) + h(n_i) = g(s)$$
$$< g(x) = g(x) + h'(x) = f(x)$$

Il teorema segue per assurdo.

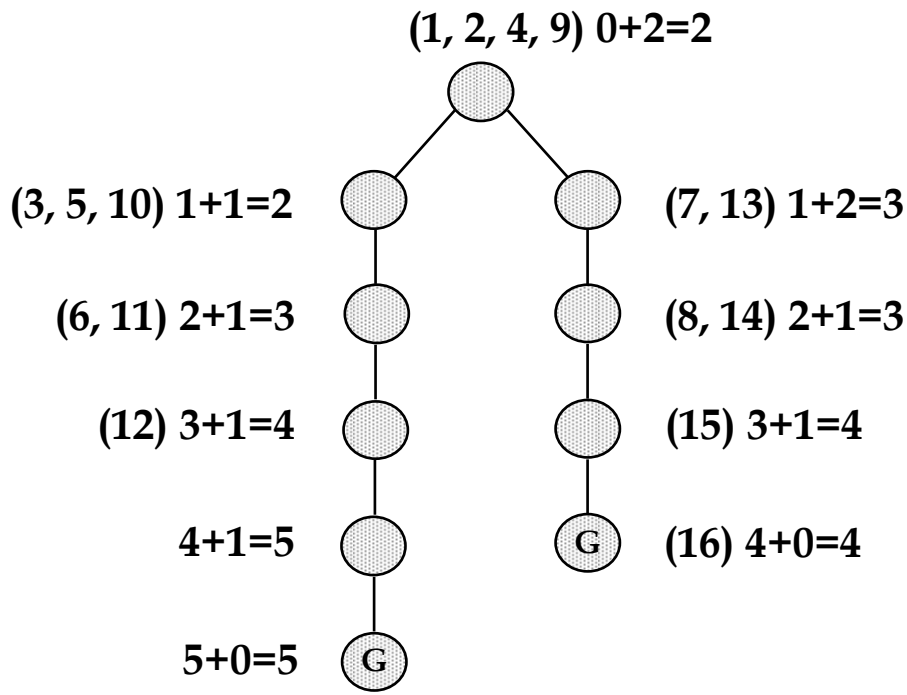
Properties of A*

- Complete? Yes (unless there are infinitely many nodes with f ≤ $f(G)$)
- Time? Exponential
- Space? Keeps all nodes in memory
- Optimal? Yes

Iterative deepening A*
Prima versione

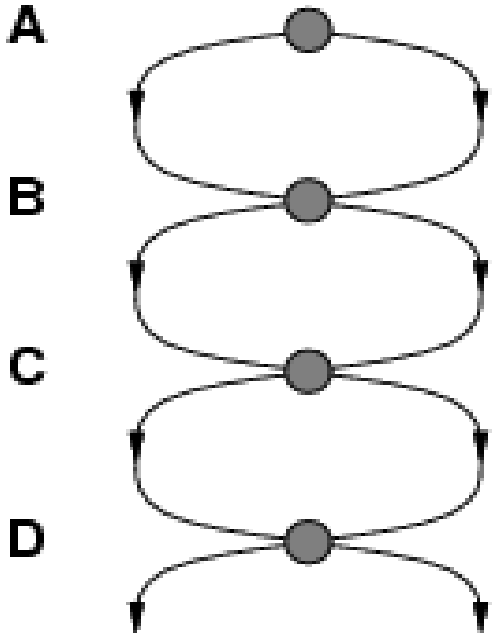
1) Poni C=1; C rappresenta il livello massimo
2) Sia *L* la lista dei nodi "iniziali"
3) Sia *n* il primo nodo di *L*. Se *L* è vuota, incrementa C e ritorna al passo 2)
4) Se *n* è un nodo "goal", allora ESCI e restituisci in output *n* ed il percorso dal nodo iniziale fino a *n*
5) Altrimenti, cancella *n* da *L*. Aggiungi all'inizio di *L* tutti i discendenti *n'* di *n* tali che $f(n') \leq C$, etichettando ciascuno di essi con il percorso dal nodo iniziale
6) Ritorna al passo 3)

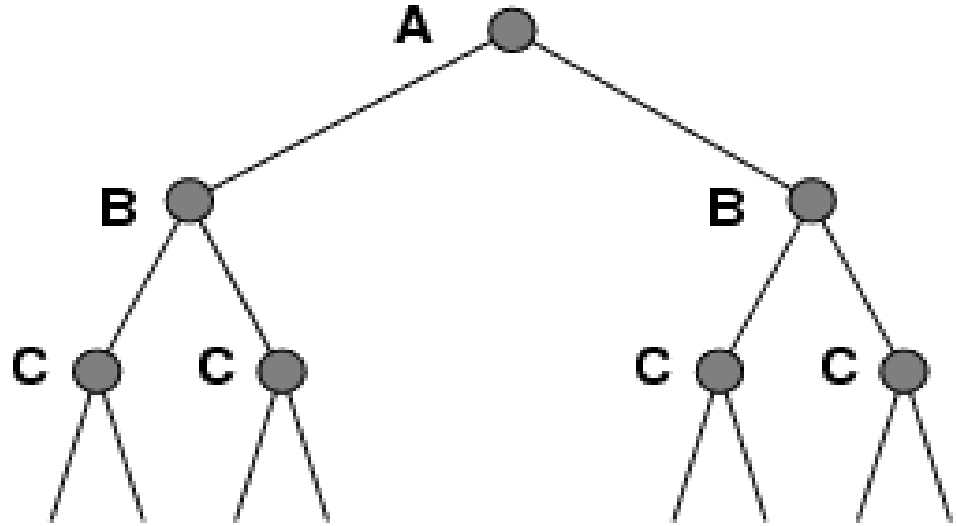
IDA*
Esempio


(1, 2, 4, 9) 0+2=2
(3, 5, 10) 1+1=2
(7, 13) 1+2=3
(6, 11) 2+1=3
(8, 14) 2+1=3
(12) 3+1=4
(15) 3+1=4
4+1=5
G
(16) 4+0=4
5+0=5
G

Repeated states

- **Failure to detect repeated states can turn a linear problem into an exponential one!**





Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Consistent heuristics

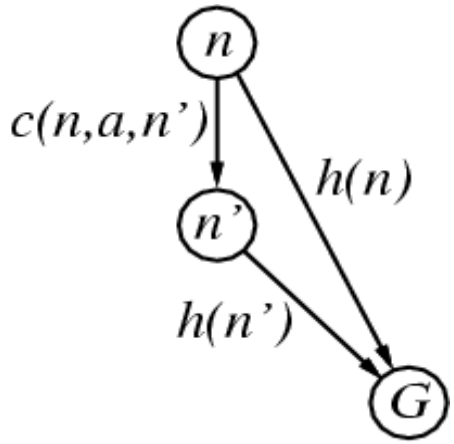
- A heuristic is consistent if for every node n, every successor n' of n generated by any action a,

  $h'(n) \leq c(n,a,n') + h'(n')$

- If h' is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h'(n') \\ &= g(n) + c(n,a,n') + h'(n') \\ &\geq g(n) + h'(n) \\ &= f(n) \end{aligned}$$

- i.e., $f(n)$ is non-decreasing along any path.

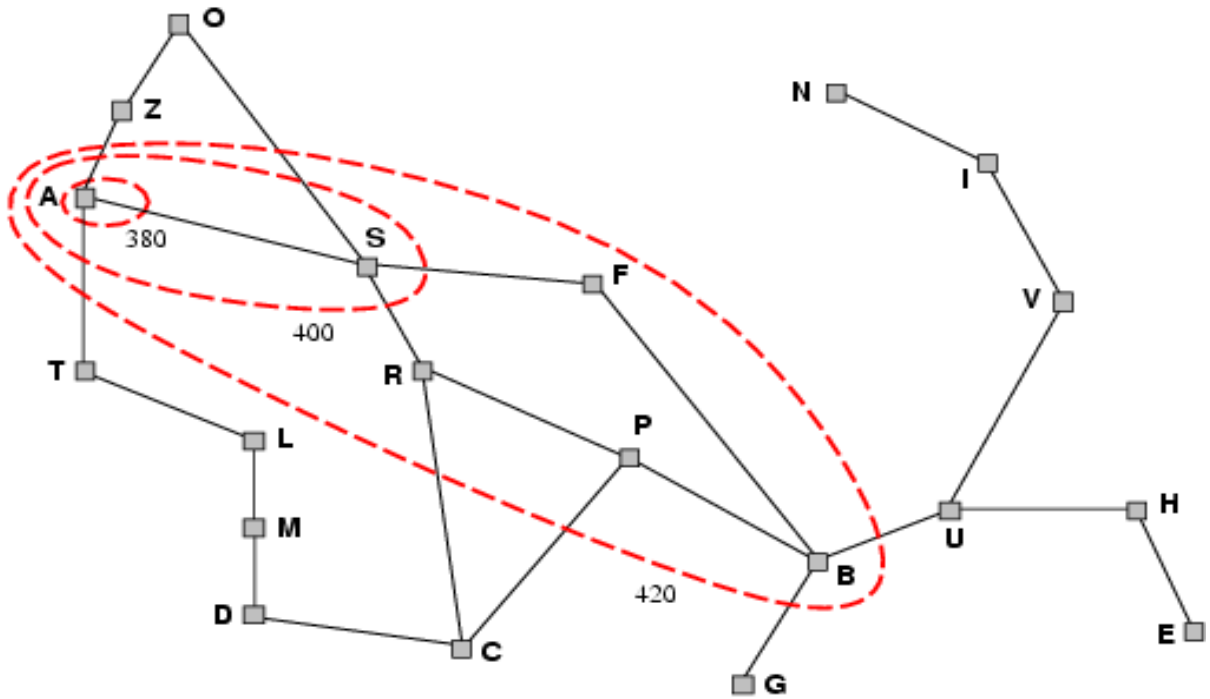


Theorem: If $h'(n)$ is consistent, A* using GRAPH-SEARCH is optimal

Optimality of A*

- **A* expands nodes in order of increasing *f* value**

- **Gradually adds "*f*-contours" of nodes**
- **Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$**



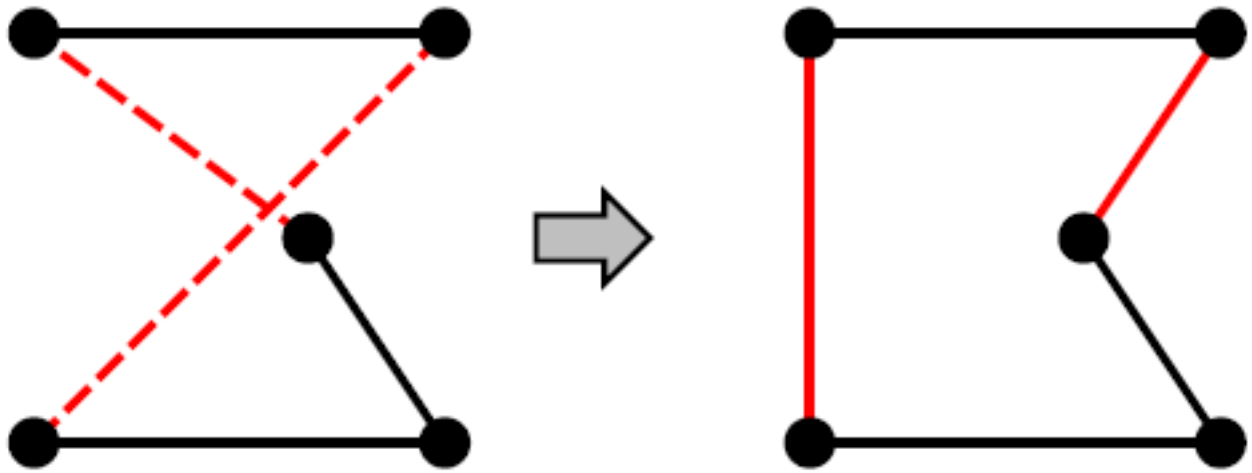
Local Search

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms keep a single "current" state, try to improve it

Example: Traveling Salesman Problem

Start with any complete tour, perform pairwise exchanges

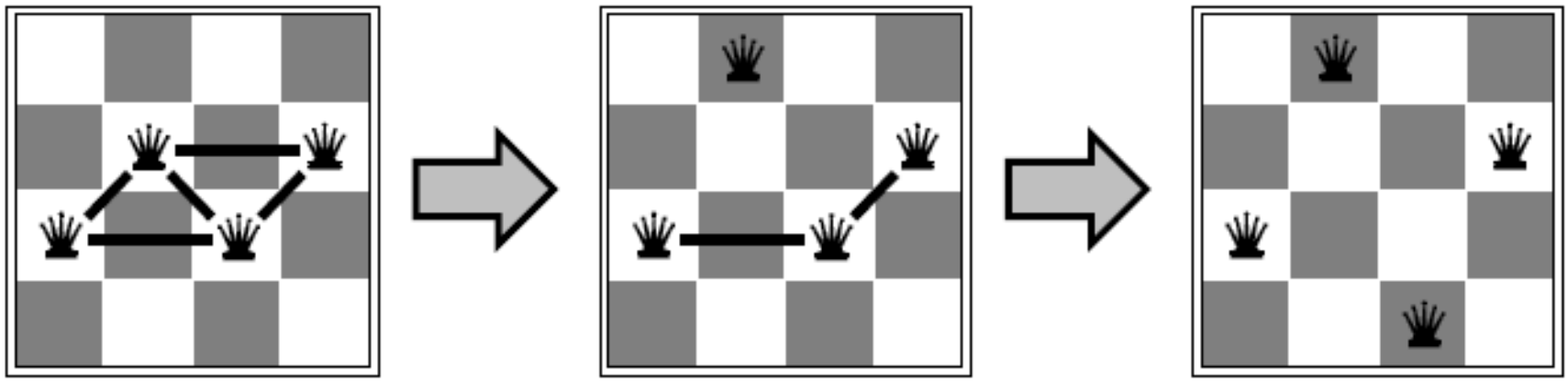


Variants of this approach get within 1% of optimal very quickly with thousands of cities

Example: *n*-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts



Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 1million$

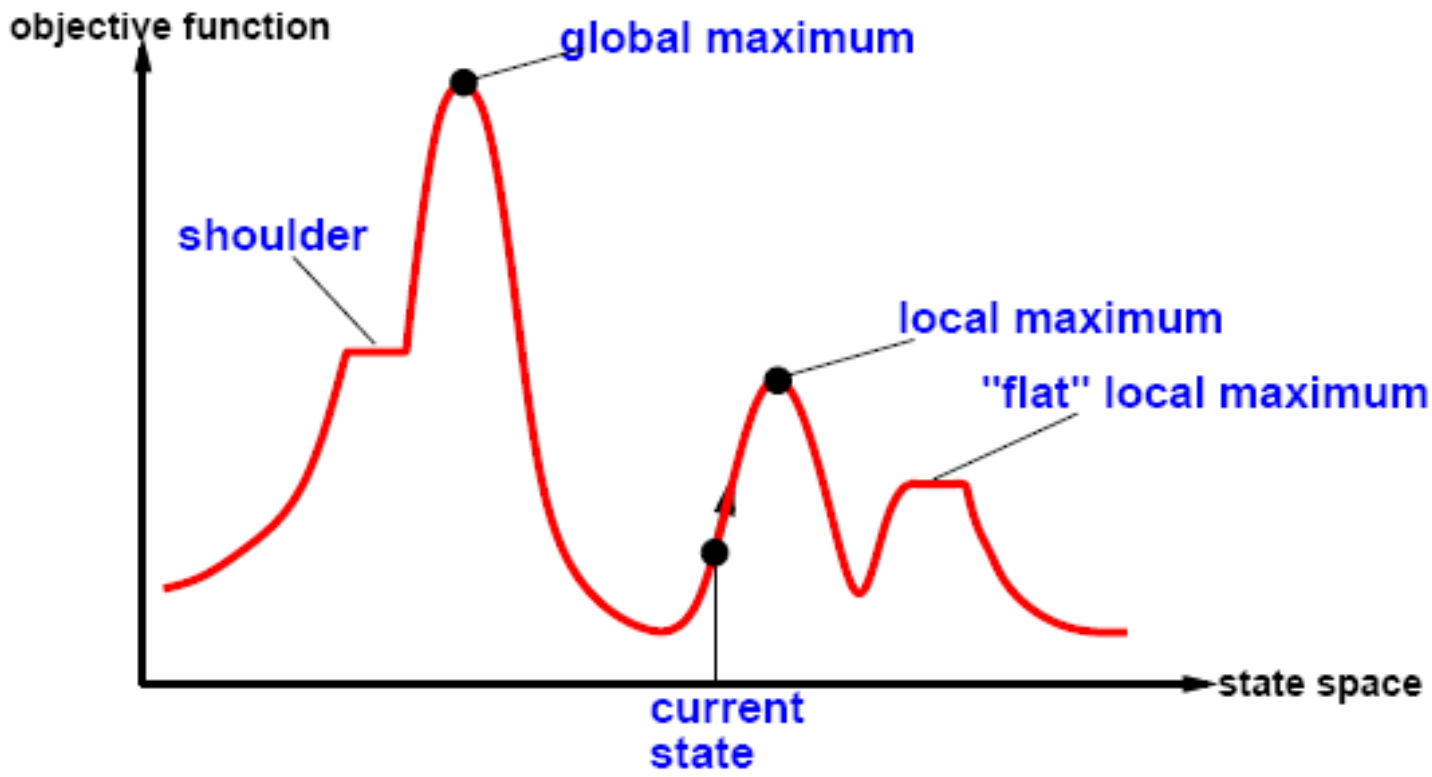
Hill-climbing search

"Like climbing Everest in thick fog with amnesia"

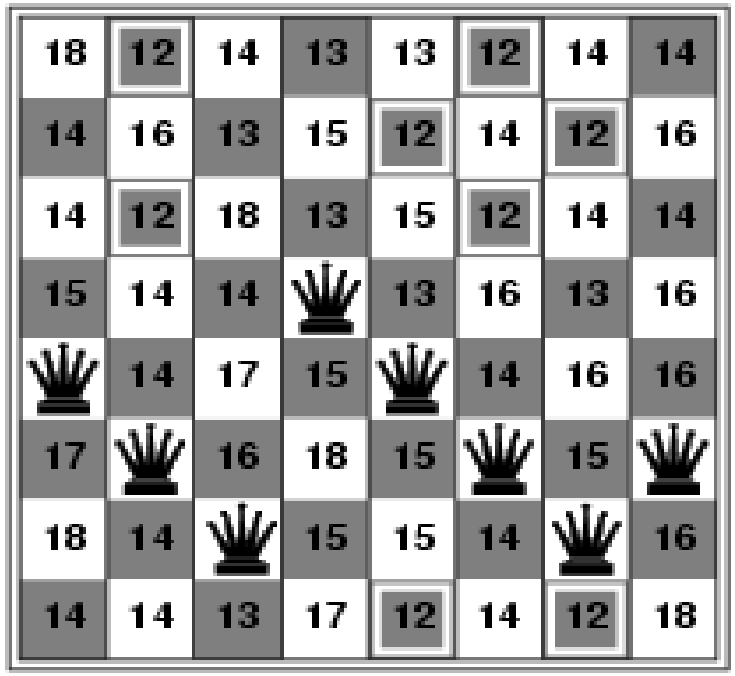
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

Hill-climbing search

- **Problem: depending on initial state, can get stuck in local maxima**

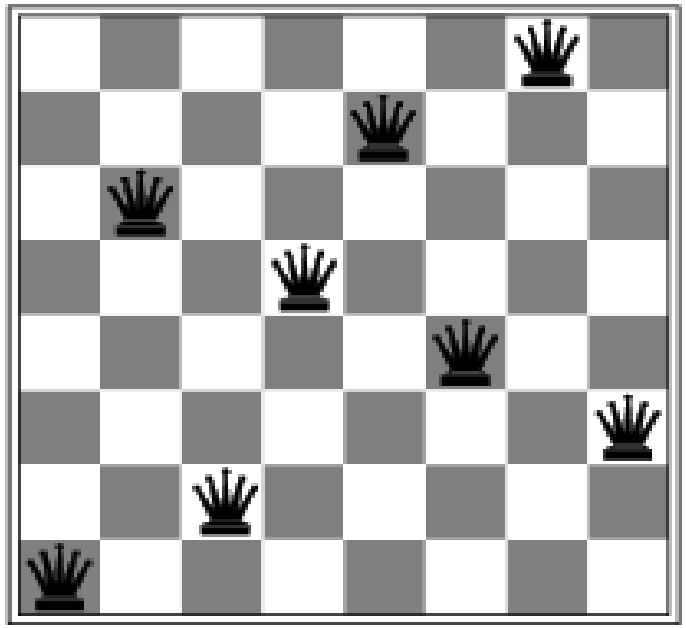


Hill-climbing search: 8-queens problem



- *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- *h* = *17* for the above state

Hill-climbing search: 8-queens problem



- A local minimum with $h = 1$

Simulated annealing search

- **Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency**

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(Δ E/T)
```

Properties of simulated annealing search

- **One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1**

- **Widely used in VLSI layout, airline scheduling, etc**

Local beam search

- **Keep track of *k* states rather than just one**
- **Start with *k* randomly generated states**
- **At each iteration, all the successors of all *k* states are generated**
- **If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.**

Genetic algorithms

- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

Cosa sono e come operano gli Algoritmi Genetici

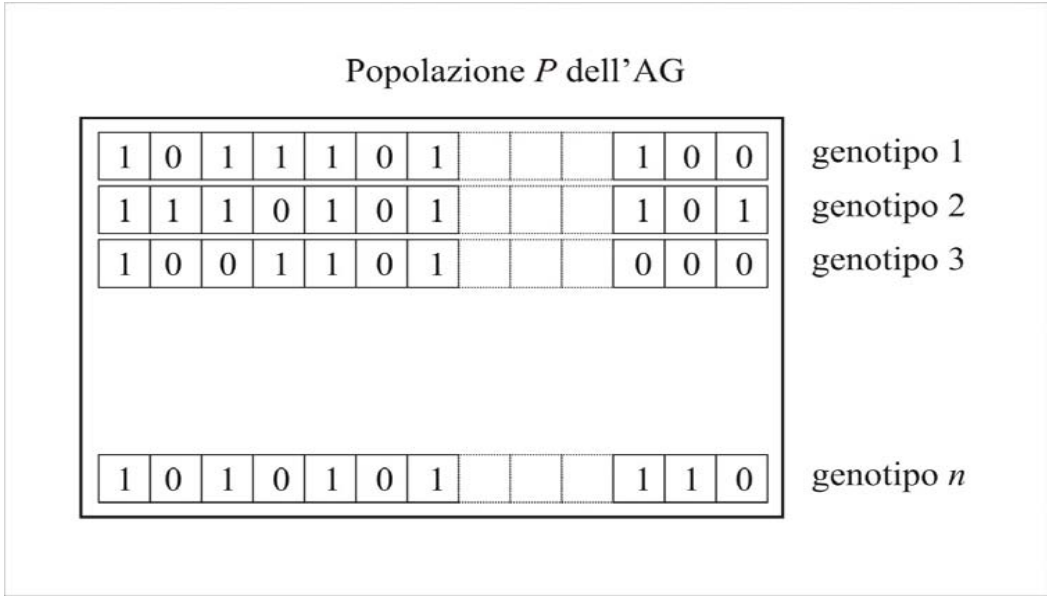
- Gli Algoritmi Genetici (AG) furono proposti da John Holland (Univeristà del Michigan) tra la fine degli anni 60 e l'inizio degli anni 70

- Gli AG (Holland, 1975, Goldberg, 1989) sono algoritmi di ricerca che si ispirano ai meccanismi della selezione naturale e della riproduzione sessuale

- Gli AG simulano l'evoluzione di una popolazione di individui, che rappresentano soluzioni candidate di uno specifico problema, favorendo la sopravvivenza e la riproduzione dei migliori

Codifica

- **L'originario modello di Holland opera su su una popolazione *P* di *n* stringhe di bit (dette individui o genotipi) di lunghezza *l* fissata**



Funzione di fitness, spazio di ricerca e "fitness landscape"

- la **funzione di fitness** valuta la bontà degli individui g_i della popolazione P nel risolvere il problema di ricerca dato:

$$f : P \rightarrow \mathrm{R}; \quad f(g_i) = f_i$$

- L'insieme delle stringhe binarie di lunghezza l ha 2^l elementi; tale insieme rappresenta lo **spazio di ricerca** (search space) dell'AG, cioè lo spazio che l'AG deve esplorare per risolvere il problema di ricerca (es. trovare il massimo o il minimo).

- I valori di fitness sui punti dello spazio di ricerca è detto **fitness landscape**.

Operatori

- **Una volta che la funzione di fitness ha determinato il valore di bontà di ogni individuo della popolazione, una nuova popolazione di individui (o genotipi) viene creata applicando alcuni operatori che si ispirano alla selezione naturale e alla genetica**

- **Gli operatori "classici" sono:**

  - **Selezione (ispirato alla selezione naturale)**
  - **Crossover (ispirato alla genetica)**
  - **Mutazione (ispirato alla genetica)**

L'operatore di selezione

- **La selezione naturale Darwiniana sostiene che gli individui più "forti" abbiano maggiori probabilità di sopravvivere nell'ambiente in cui vivono e, dunque, maggiore probabilità di riprodursi**

- **Nel contesto dell'AG di Holland, gli individui più forti sono quelli con fitness più alta, poiché risolvono meglio di altri il problema di ricerca dato; per questo essi devono essere privilegiati nella fase di selezione di quegli individui che potranno riprodursi dando luogo a nuovi individui**

La selezione proporzionale (roulette wheel selection)

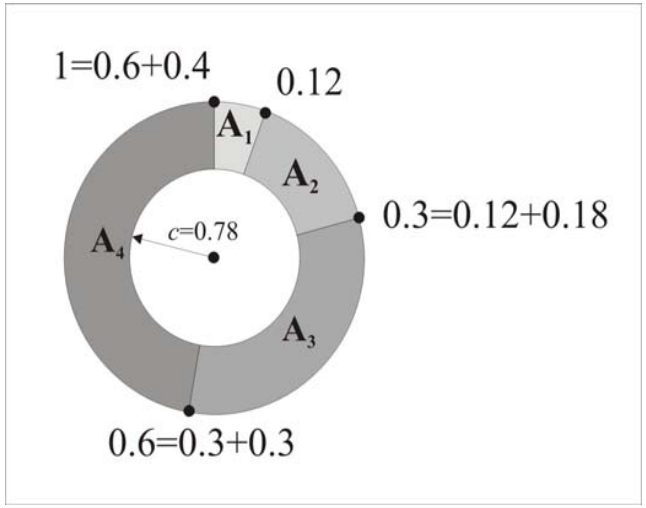
- **Metodo di selezione proporzionale al valore di fitness;**

- **Sia f_i il valore di fitness del genotipo g_i, la probabilità che g_i sia selezionato per la riproduzione è:**

$$p_i = f_i \Big/ \sum_k f_k$$

- **Tali probabilità sono utilizzate per costruire una sorta di roulette**

Esempio di roulette



- **I quattro individui A_1, A_2, A_3 e A_4, con probabilità di selezione 0.12, 0.18, 0.3 e 0.4, occupano uno "spicchio" di roulette di ampiezza pari alla propria probabilità di selezione.**

- **Nell'esempio l'operatore di selezione genera il numero casuale *c* = 0.78 e l'individuo A_4 viene selezionato**

Mating pool

- **Ogni volta che un individuo della popolazione è selezionato ne viene creata una copia; tale copia è inserita nel così detto mating pool**

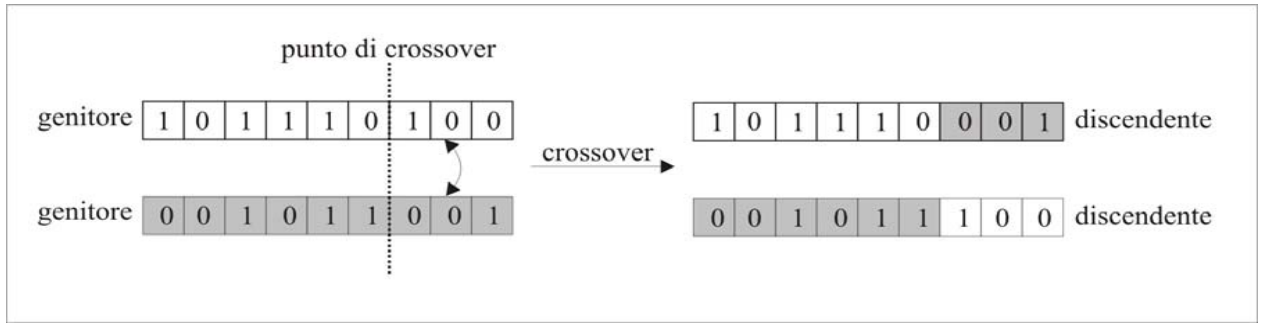
- **Quando il mating pool è riempito con esattamente *n* (numero di individui della popolazione) copie di individui della popolazione, nuovi *n* discendenti sono creati applicando gli operatori genetici**

Crossover

- **Si scelgono a caso due individui nel mating pool (genitori) e un punto di taglio (punto di crossover) su di essi. Le porzioni di genotipo alla destra del punto di crossover vengono scambiate generando due discendenti.**

- **L'operatore di crossover è applicato, in accordo a una prefissata probabilità p_c, $n/2$ volte in modo da ottenere n discendenti;**

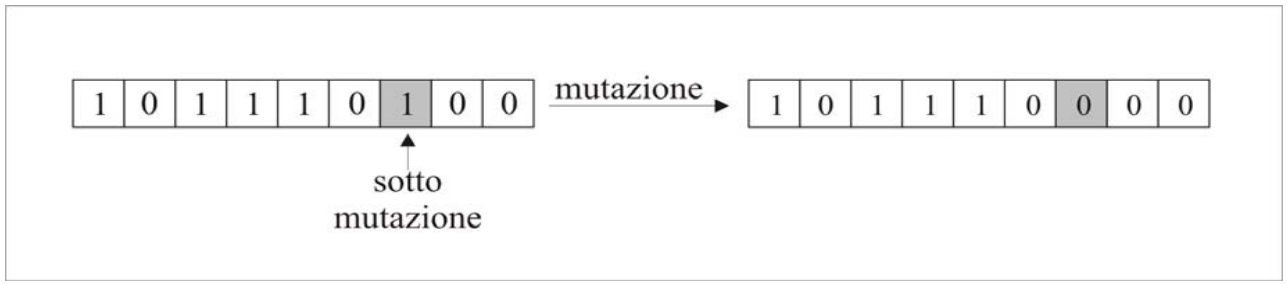
- **Nel caso in cui il crossover non sia applicato, i discendenti coincidono con i genitori.**



Mutazione

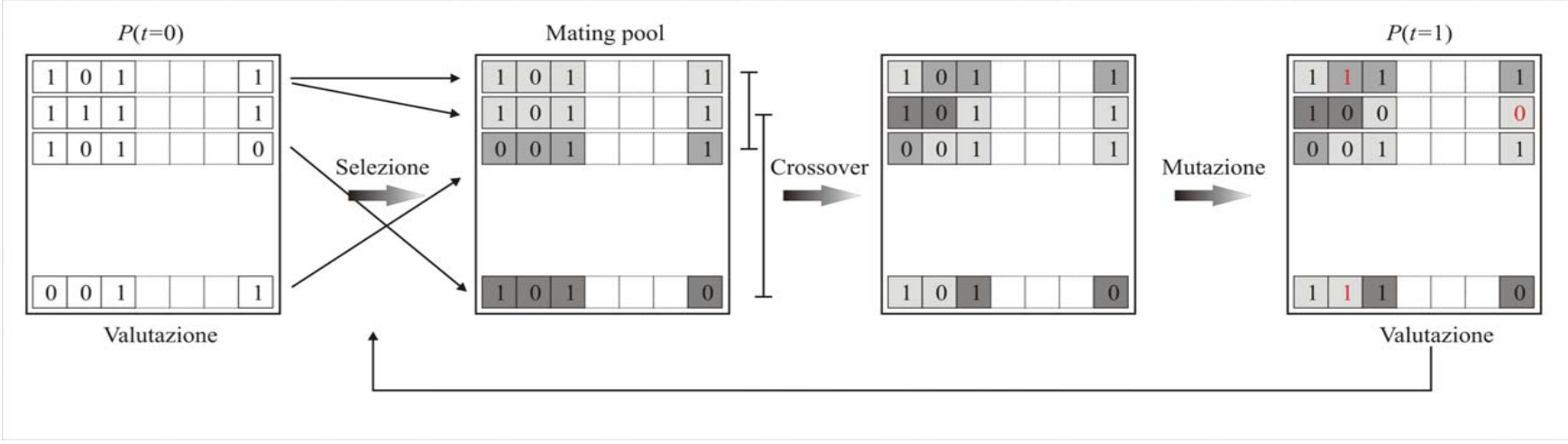
- **Una volta che due discendenti sono stati generati per mezzo del crossover, il valore dei bit dei nuovi individui sono cambiati da 0 in 1 o viceversa in funzione di una (tipicamente piccola) probabilita p_m, .**

- **Come il crossover rappresenta una metafora della riproduzione sessuale, l'operatore di mutazione modella il fenomeno genetico della rara variazione di elementi del genotipo negli esseri viventi durante la riproduzione sessuale.**

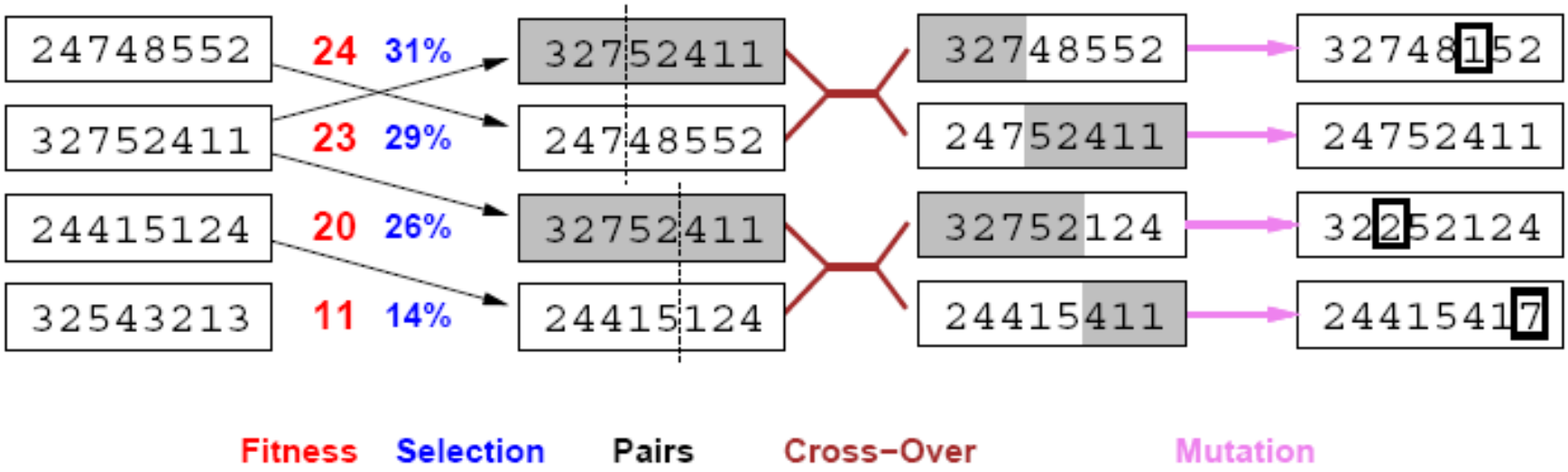


L'algoritmo (Simple Genetic Algorithm)

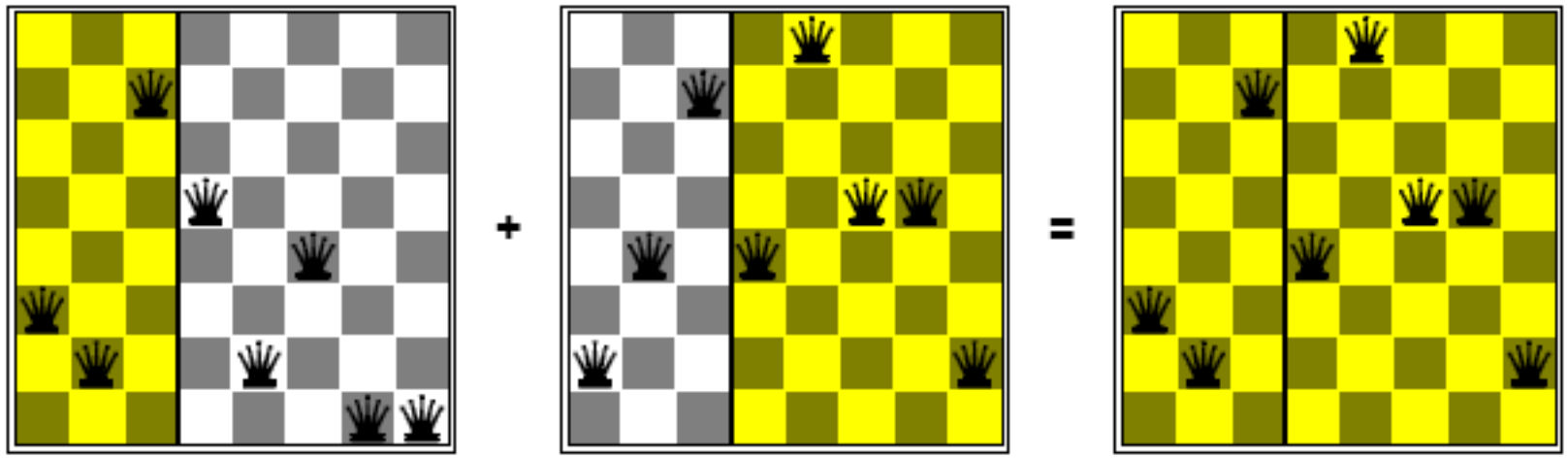
```
SGA {
        t=0
        inizializza la popolazione P(t) in maniera casuale
        valuta la popolazione P(t)
        mentre (!criterio_fermata) {
                t=t+1
                crea P(t) applicando selezione, crossover e mutazione
                valuta P(t)
        }
}
```



Example: *n*-queens



- **Fitness function: number of *non-attacking* pairs of queens (min = 0, max = 8 × 7/2 = 28)**
- **24/(24+23+20+11) = 31%**
- **23/(24+23+20+11) = 29% etc**



Search in Continuous State Spaces *

* From D. Luenberger, *Linear and Nonlinear Programming*.

Optimization problems

The general mathematical programming problem can be stated as

$$
\begin{aligned}
\text{minimize} \quad & f(\mathbf{x}) \\
\text{subject to} \quad & h_i(\mathbf{x}) = 0, \qquad i = 1, 2, \ldots, m \\
& g_j(\mathbf{x}) \leq 0, \qquad j = 1, 2, \ldots r \\
& \mathbf{x} \in S.
\end{aligned}
$$

In this formulation, $\mathbf{x}$ is an n-dimensional vector of unknowns, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, and f, h_i, $i = 1, 2, \ldots, m$, and g_j, $j = 1, 2, \ldots, r$, are real-valued functions of the variables $x_1, x_2, \ldots, x_n$. The set S is a subset of n-dimensional space. The function f is the *objective function* of the problem and the equations, inequalities, and set restrictions are *constraints*.

In this chapter we consider optimization problems of the form

$$\begin{aligned} &\text{minimize} && f(\mathbf{x}) \\ &\text{subject to} && \mathbf{x} \in \Omega, \end{aligned} \tag{1}$$

where f is a real-valued function and Ω, the feasible set, is a subset of E^n.

In an investigation of the general problem (1) we distinguish two kinds of solution points: *local minimum points*, and *global minimum points*.

Definition. A point $\mathbf{x}^* \in \Omega$ is said to be a *relative minimum point* or a *local minimum point* of f over Ω if there is an $\varepsilon > 0$ such that $f(\mathbf{x}) \geqslant f(\mathbf{x}^*)$ for all $\mathbf{x} \in \Omega$ within a distance ε of $\mathbf{x}^*$ (that is, $\mathbf{x} \in \Omega$ and $|\mathbf{x} - \mathbf{x}^*| < \varepsilon$). If $f(\mathbf{x}) > f(\mathbf{x}^*)$ for all $\mathbf{x} \in \Omega$, $\mathbf{x} \neq \mathbf{x}^*$, within a distance ε of $\mathbf{x}^*$, then $\mathbf{x}^*$ is said to be a *strict relative minimum point* of f over Ω.

Definition. A point $\mathbf{x}^* \in \Omega$ is said to be a *global minimum point* of f over Ω if $f(\mathbf{x}) \geqslant f(\mathbf{x}^*)$ for all $\mathbf{x} \in \Omega$. If $f(\mathbf{x}) > f(\mathbf{x}^*)$ for all $\mathbf{x} \in \Omega$, $\mathbf{x} \neq \mathbf{x}^*$, then $\mathbf{x}^*$ is said to be a *strict global minimum point* of f over Ω.

Types of optimization problems

- Linear versus nonlinear optimization

- Unconstrained versus constrained optimization

Linear programming

$$\begin{aligned} &\text{minimize} && \mathbf{c}^T\mathbf{x} \\ &\text{subject to} && \mathbf{A}\mathbf{x} = \mathbf{b} \quad \text{and} \quad \mathbf{x} \geqslant \mathbf{0}. \end{aligned}$$

Algorithms:

- simplex method
- interior-point methods

Unconstrained problems: The method of gradient descent

Let f have continuous first partial derivatives on E^n. We will frequently have need for the gradient vector of f and therefore we introduce some simplifying notation. The gradient $\nabla f(\mathbf{x})$ is, according to our conventions, defined as a n-dimensional *row* vector. For convenience we define the n-dimensional *column* vector $\mathbf{g}(\mathbf{x}) = \nabla f(\mathbf{x})^T$. When there is no chance for ambiguity, we sometimes suppress the argument $\mathbf{x}$ and, for example, write $\mathbf{g}_k$ for $\mathbf{g}(\mathbf{x}_k) = \nabla f(\mathbf{x}_k)^T$.

The method of steepest descent is defined by the iterative algorithm

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k,$$

where α_k is a nonnegative scalar minimizing $f(\mathbf{x}_k - \alpha \mathbf{g}_k)$. In words, from the point $\mathbf{x}_k$ we search along the direction of the negative gradient $-\mathbf{g}_k$ to a minimum point on this line; this minimum point is taken to be $\mathbf{x}_{k+1}$.

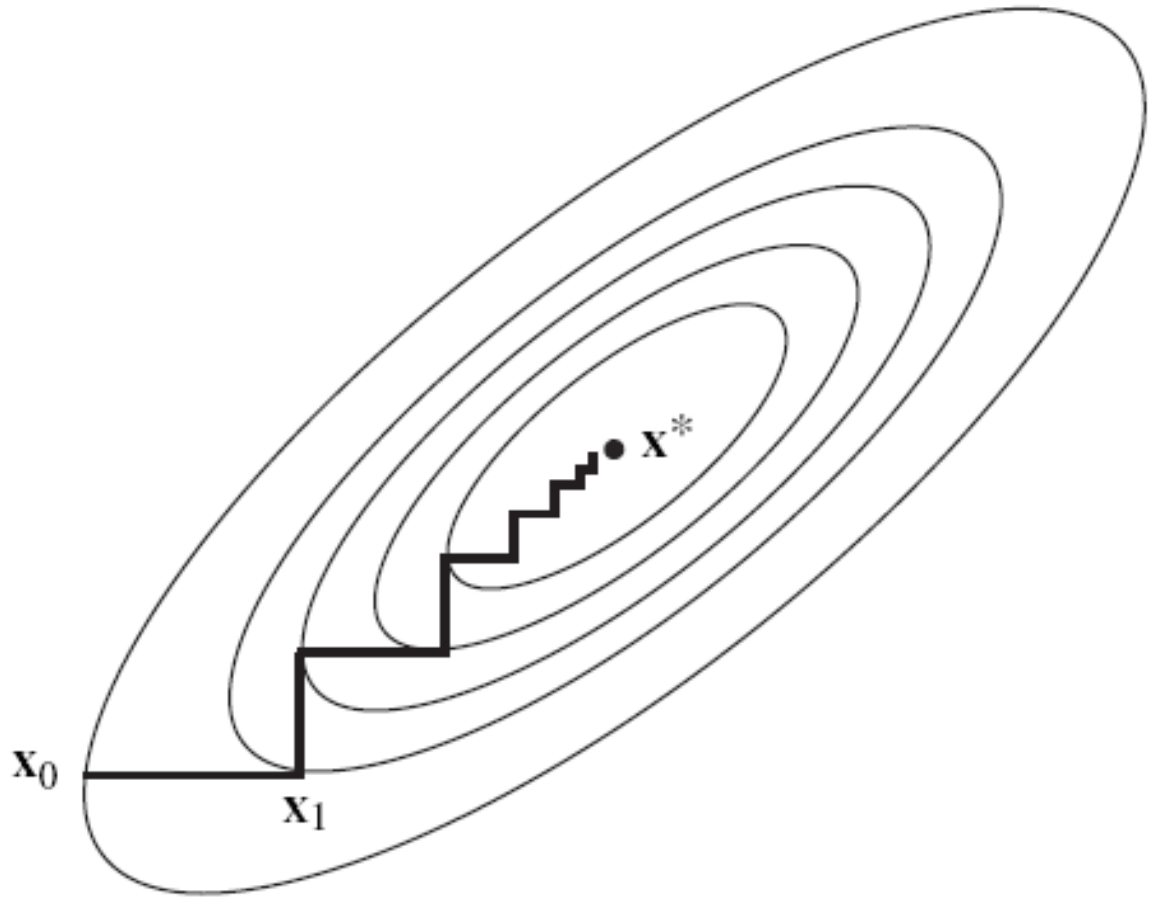


Fig. 8.9 Steepest descent

Unconstrained problems: Newton's method

The idea behind Newton's method is that the function f being minimized is approximated locally by a quadratic function, and this approximate function is minimized exactly. Thus near $\mathbf{x}_k$ we can approximate f by the truncated Taylor series

$$f(\mathbf{x}) \simeq f(\mathbf{x}_k) + \boldsymbol{\nabla} f(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) + \tfrac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T \mathbf{F}(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k).$$

The right-hand side is minimized at

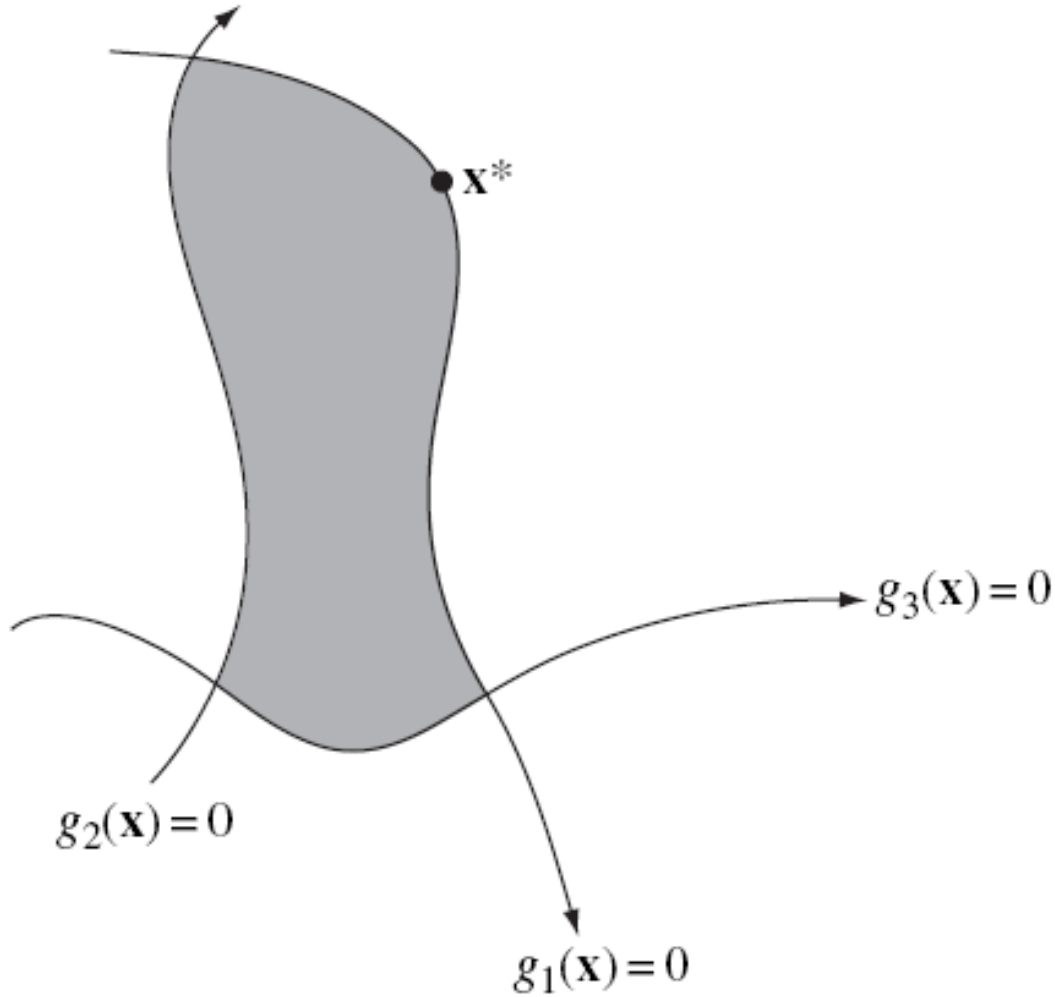
$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{F}(\mathbf{x}_k)]^{-1} \boldsymbol{\nabla} f(\mathbf{x}_k)^T, \tag{48}$$

and this equation is the pure form of Newton's method.

Constrained optimization

We deal with general nonlinear programming problems of the form

$$
\begin{array}{lll}
\text{minimize} & f(\mathbf{x}) & \\
\text{subject to} & h_1(\mathbf{x}) = 0 & g_1(\mathbf{x}) \leqslant 0 \\
& h_2(\mathbf{x}) = 0 & g_2(\mathbf{x}) \leqslant 0 \\
& \vdots & \vdots \\
& h_m(\mathbf{x}) = 0 & g_p(\mathbf{x}) \leqslant 0 \\
& \mathbf{x}, \in \Omega \subset E^n, &
\end{array}
\tag{1}
$$


$\mathbf{x}^*$
$g_3(\mathbf{x}) = 0$
$g_2(\mathbf{x}) = 0$
$g_1(\mathbf{x}) = 0$

in vectorial notations…

$$\begin{array}{ll} \text{minimize} & f(\mathbf{x}) \\ \text{subject to} & \mathbf{h}(\mathbf{x}) = \mathbf{0},\ \mathbf{g}(\mathbf{x}) \leqslant \mathbf{0} \\ & \mathbf{x} \in \Omega. \end{array} \tag{2}$$

The constraints $\mathbf{h}(\mathbf{x}) = \mathbf{0}, \mathbf{g}(\mathbf{x}) \leqslant \mathbf{0}$ are referred to as *functional constraints*, while the constraint $\mathbf{x} \in \Omega$ is a *set constraint*. As before we continue to de-emphasize the set constraint, assuming in most cases that either Ω is the whole space E^n or that the solution to (2) is in the interior of Ω. A point $\mathbf{x} \in \Omega$ that satisfies all the functional constraints is said to be *feasible*.

The Lagrangian: The equality constraint case

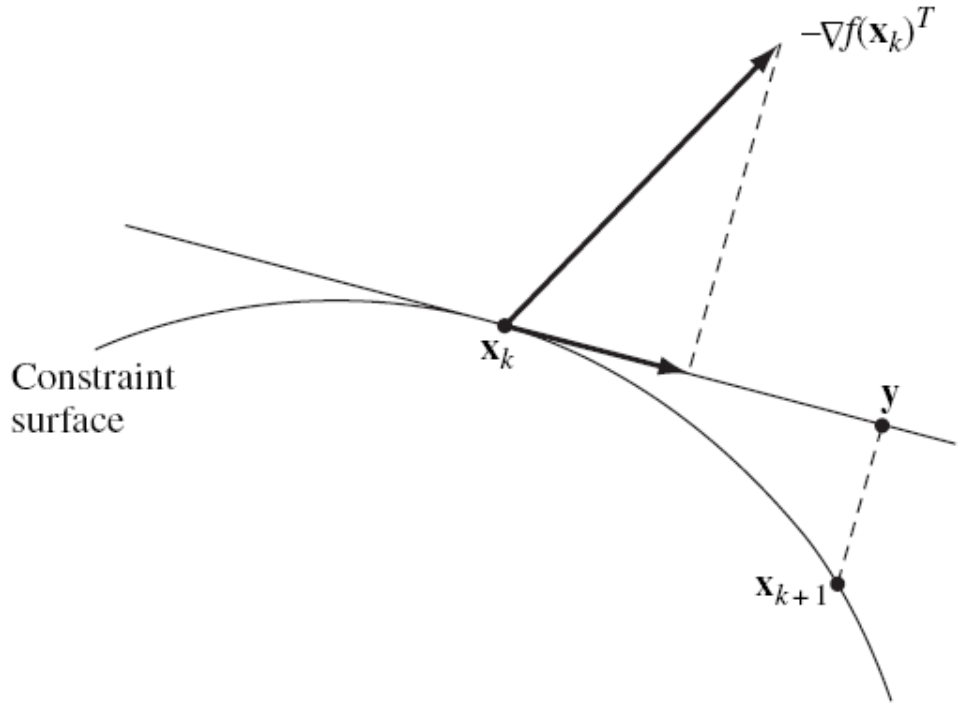
$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{aligned}$$

Theorem. *Let* $\mathbf{x}^*$ *be a local extremum point of* f *subject to the constraints* $\mathbf{h}(\mathbf{x}) = \mathbf{0}$. *Assume further that* $\mathbf{x}^*$ *is a regular point of these constraints. Then there is a* $\boldsymbol{\lambda} \in E^m$ *such that*

$$\nabla f(\mathbf{x}^*) + \boldsymbol{\lambda}^T \nabla \mathbf{h}(\mathbf{x}^*) = \mathbf{0}. \tag{7}$$

The gradient projection method

The gradient projection method is motivated by the ordinary method of gradient descent for unconstrained problems. The negative gradient is projected onto the working surface in order to define the direction of movement



Penalty methods

Penalty methods approximate a constrained problem by an unconstrained problem that assigns high cost to points that are far from the feasible region.

As the approximation is made more exact (by letting the parameter c tend to infinity) the solution of the unconstrained penalty problem approaches the solution to the original constrained problem from outside the active constraints.

Penalty methods

Consider the problem

$$\begin{aligned} &\text{minimize} \quad f(\mathbf{x}) \\ &\text{subject to} \quad \mathbf{x} \in S, \end{aligned} \tag{1}$$

where f is a continuous function on E^n and S is a constraint set in E^n. In most applications S is defined implicitly by a number of functional constraints, but in this section the more general description in (1) can be handled. The idea of a penalty function method is to replace problem (1) by an unconstrained problem of the form

$$\text{minimize} \quad f(\mathbf{x}) + cP(\mathbf{x}), \tag{2}$$

where c is a positive constant and P is a function on E^n satisfying: (i) P is continuous, (ii) $P(\mathbf{x}) \geqslant 0$ for all $\mathbf{x} \in E^n$, and (iii) $P(\mathbf{x}) = 0$ if and only if $\mathbf{x} \in S$.

Example 1. Suppose S is defined by a number of inequality constraints:

$$S = \{\mathbf{x} : g_i(\mathbf{x}) \leqslant 0, \quad i = 1, 2, \ldots, p\}.$$

A very useful penalty function in this case is

$$P(\mathbf{x}) = \tfrac{1}{2} \sum_{i=1}^{P} (\max\,[0, g_i(\mathbf{x})])^2.$$