POLITECNICO DI MILANO

Algoritmi di ricerca locale

Algoritmi, Strutture Dati e Calcolo Parallelo

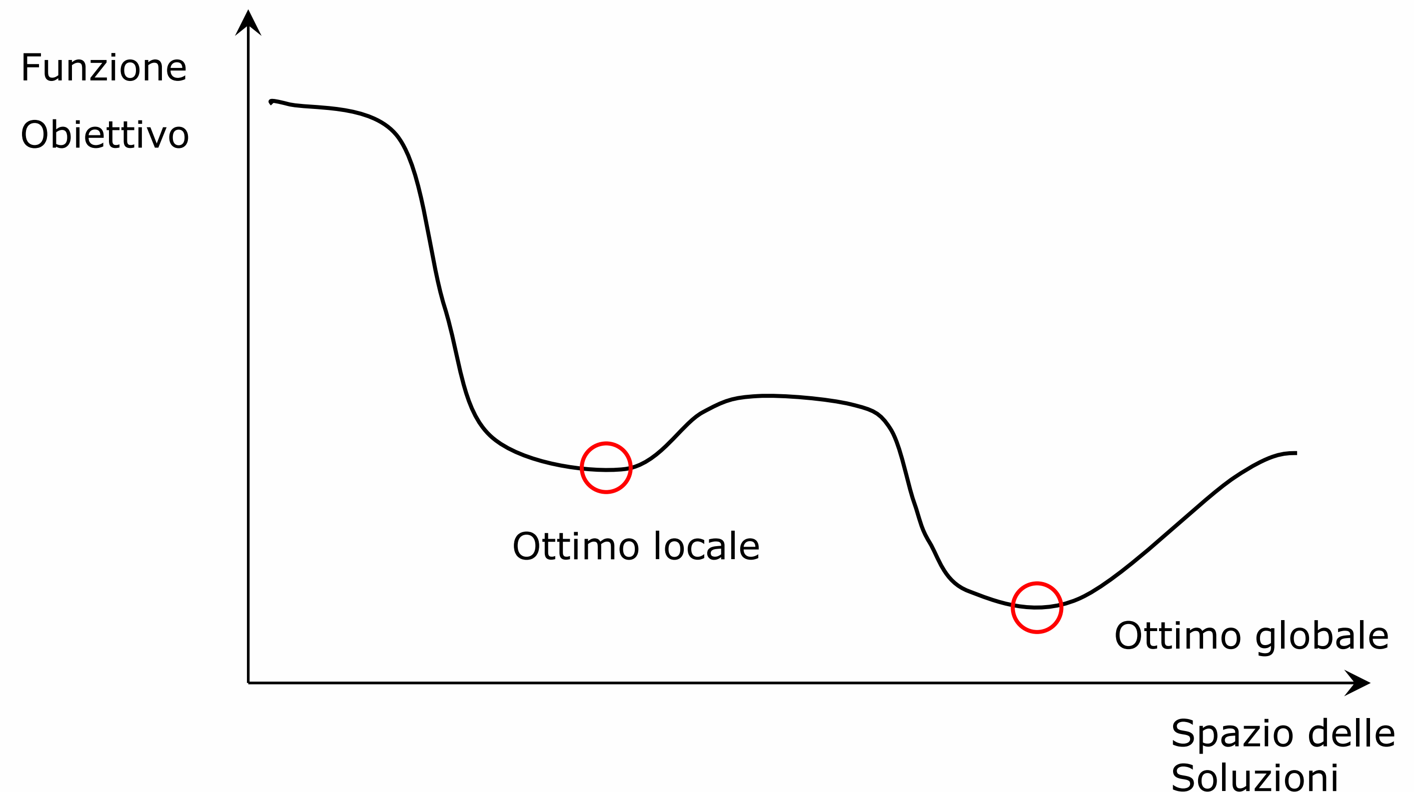
Daniele Loiacono

Quando si applicano?

- Molti problemi di ottimizzazione non possono essere risolti ne con un approccio greedy ne con metodi esatti
- I metodi esatti potrebbero non essere applicabili
  - non è possibile formalizzare opportunamente il problema
  - troppo costosi computazionalmente
- In questi casi un algoritmo euristico di ricerca locale può essere la scelta migliora se:
  - il problema ammette la definizione di intorno: data una soluzione **s**, esiste un insieme di soluzioni **N(s)** che rappresentano tutte le soluzioni raggiungibili da **s** con una sola *mossa*
  - non si è interessati al *percorso* che porta alla soluzione (ma soltanto alla soluzione stessa)
  - disponibile una funzione di valutazione (o obiettivo) **z(s)** che permette di calcolare quantitativamente il costo della soluzione

Overview

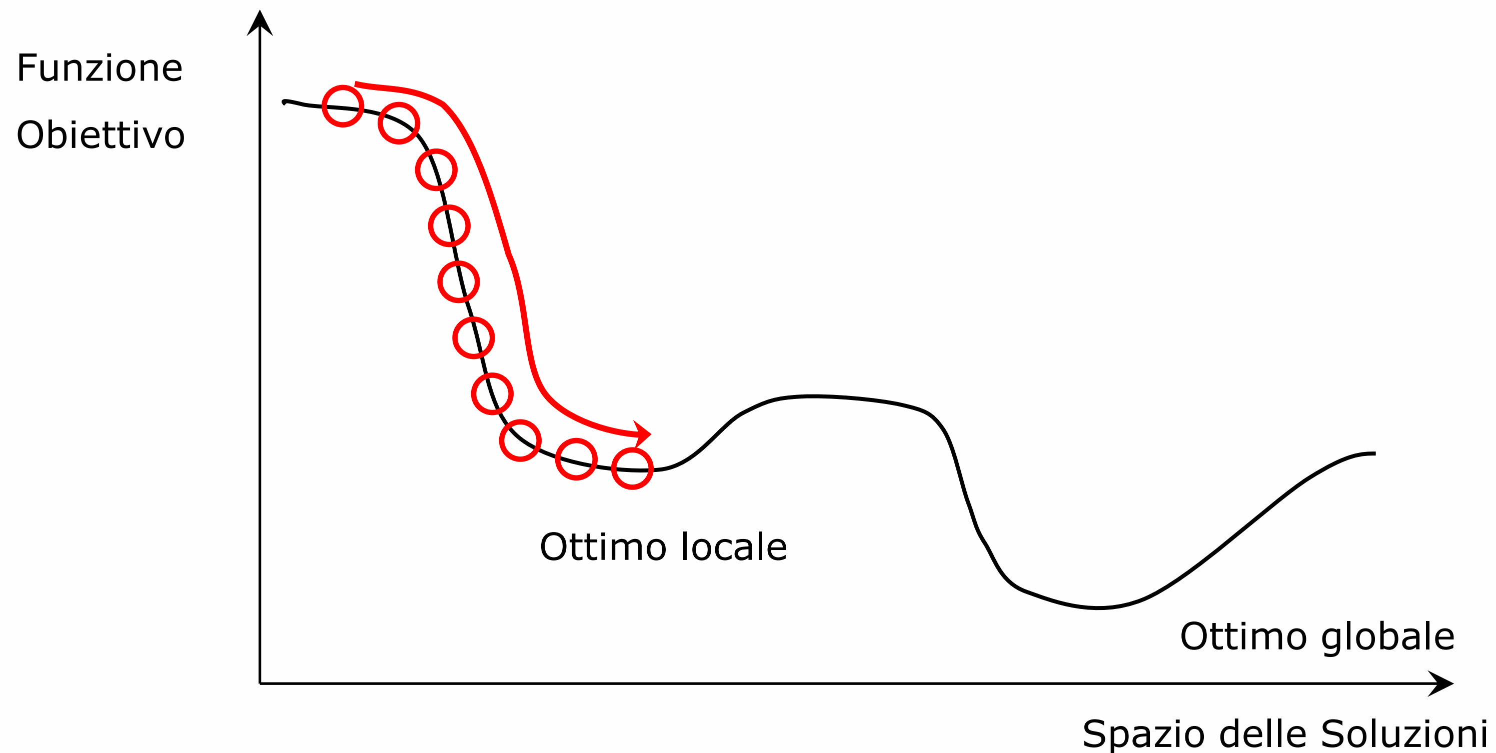
- Sono algoritmi di ottimizzazione basati su una ricerca locale
  - Partono da una soluzione e ne esplorano l'intorno
  - Si focalizzano sulla soluzione più promettente nell'intorno
- Sono necessari meccanismi specifici per evitare di rimanere bloccati in ottimi locali
- Metodi tipici di ricerca locale sono:
  - Hill Climber
  - Simulated Annealing
  - Local Beam Search
  - Tabu Search



Hill Climber

Overview

- Analizza l’intorno della soluzione corrente e si sposta in quella migliore
- È l’algoritmo di ricerca locale più semplice ma ha grosse limitazioni:
  - Non è in grado di evitare ottimi locali
  - La soluzione a cui converge dipende dalla soluzione di partenza
- Permette di ottenere buoni risultati solo in problemi molto semplici



Algoritmo Base


```
genera una soluzione iniziale s di costo z(s)
while (not CRITERIO_TERMINAZIONE) {
   genera l'intorno N(s)
   trova la migliore soluzione s' ∈ N(s) rispetto a z(·)
   if (z(s') < z (s)) {
        s = s'
   else
        TERMINA_RICERCA
   }
}
s*=s /*salva la migliore soluzione*/
```

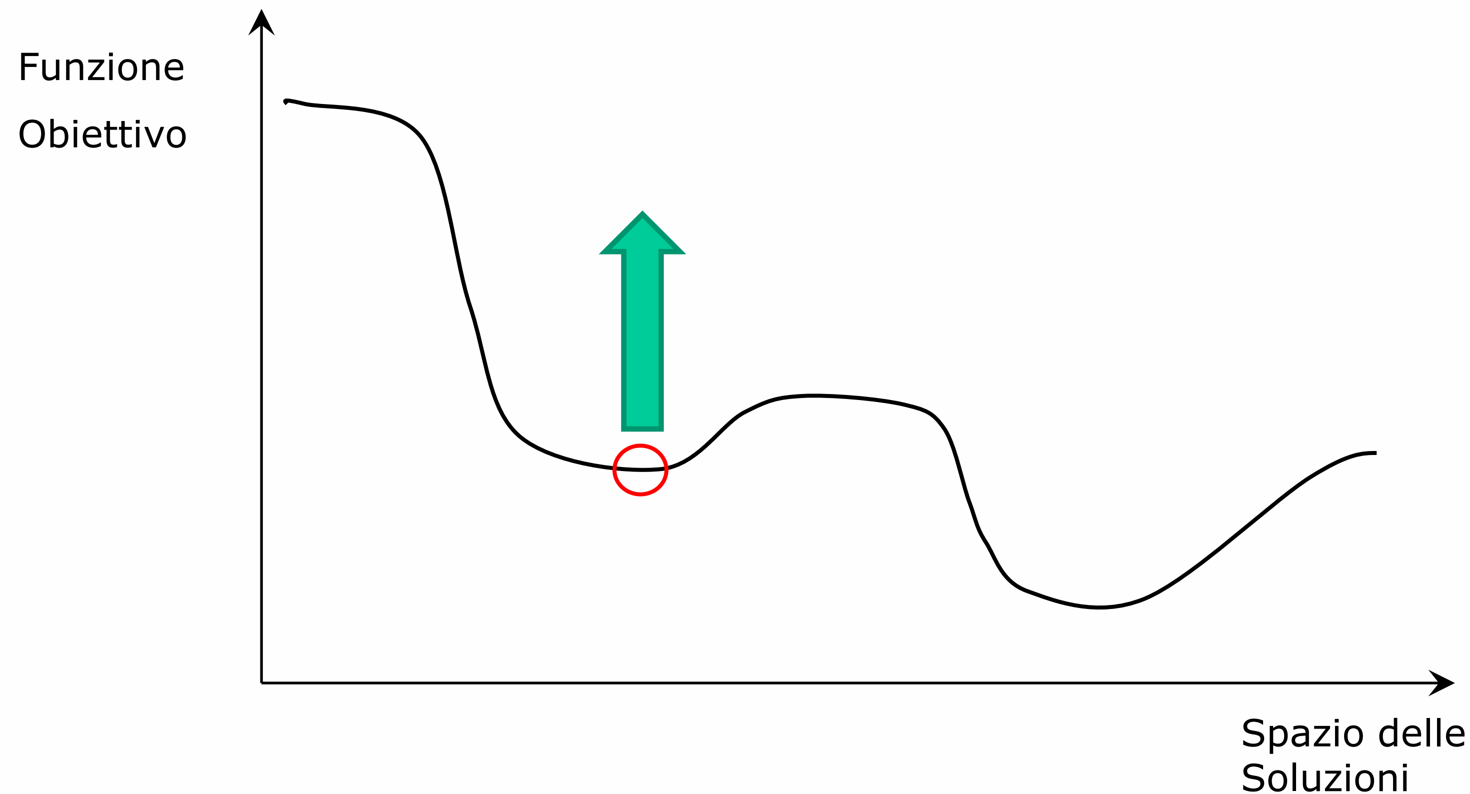
- **Stochastic**
  - Seleziona il successore (soluzione successiva nell'intorno) in maniera probabilistica (purchè sia migliore della soluzione corrente)
  - La probabilità di scelta può dipendere dall'entità del miglioramento
  - Più lento ma può portare a soluzioni migliori
- **First-choice**
  - Seleziona il primo successore generato che risulta migliore della soluzione corrente
  - Utile nei problemi in cui l'intorno è molto grande
- **Random-restart**
  - Ripete diverse volte l'algoritmo base partendo da diverse soluzioni generate casualmente
  - In alcuni problemi può essere sufficiente per evitare ottimi locali

Simulated Annealing

- Il Simulated Annealing (SA) sceglie una nuova soluzione nell'intorno di quella corrente in maniera probabilistica.
- Se la soluzione trovata è migliore di quella corrente, viene sempre accettata come nuova soluzione corrente ed il processo continua
- Se la soluzione è peggiore, viene accettata con probabilità pari $e^{\Delta z/T(i)}$
  - **Δz** è la differenza fra il costo delle due soluzioni (sempre <0)
  - **T(i)** è una funzione che decresce al crescere dell'iterazione **i** dell'algoritmo
- Questo schema probabilistico ha due proprietà:
  - La scelta di una soluzione peggiorativa diventa sempre meno probabile all'aumentare del numero di iterazioni **i**
  - Più una soluzione è peggiore della corrente, minore è la probabilità che venga scelta
- Il SA è stato molto usato in diversi ambiti applicativi (e.g., design VLSI)

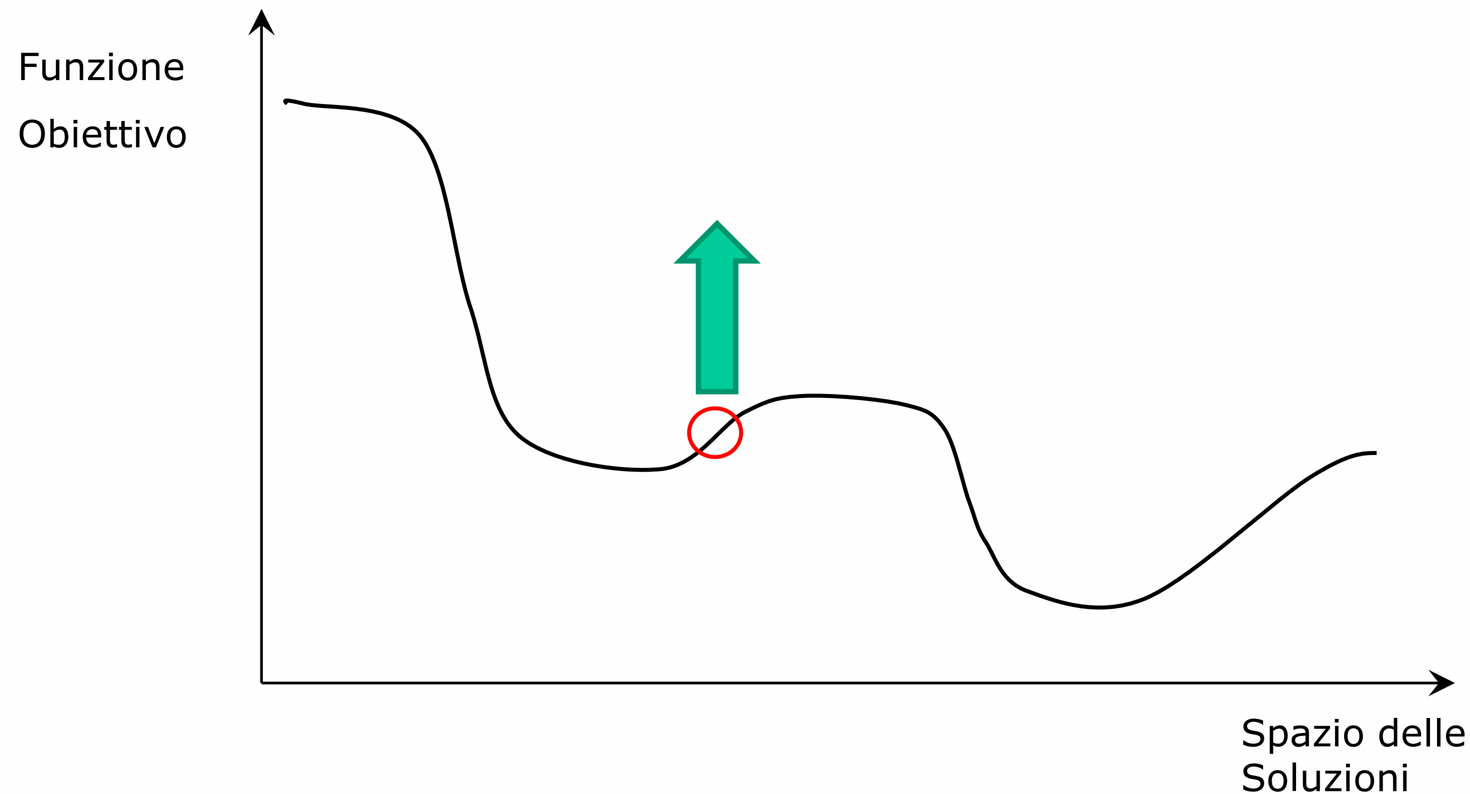
Overview (2)

- Grazie alla possibilità di poter scegliere soluzioni peggiorative, il SA è più robusto agli ottimi locali rispetto ad un hill climber



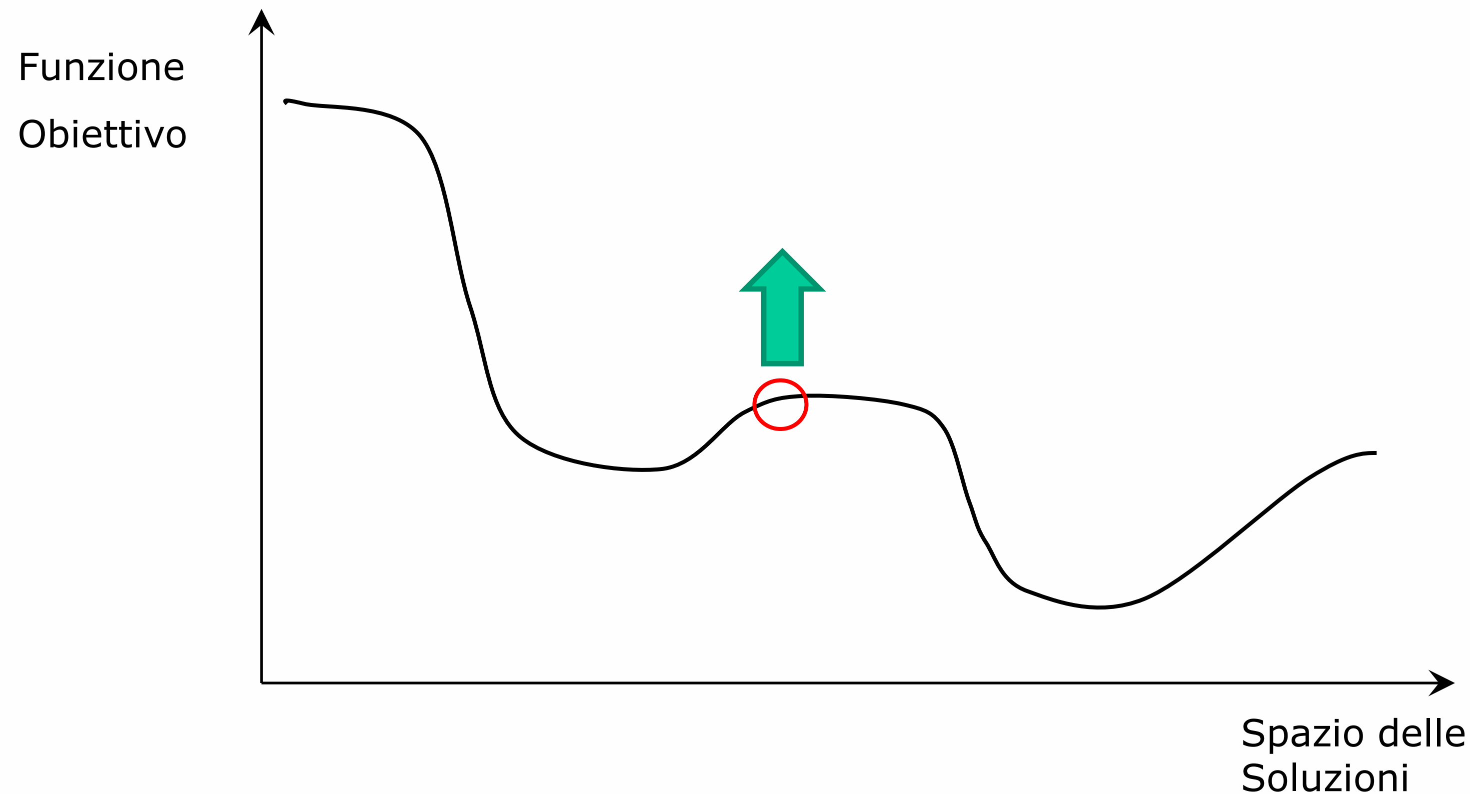
Overview (2)

- Grazie alla possibilità di poter scegliere soluzioni peggiorative, il SA è più robusto agli ottimi locali rispetto ad un hill climber



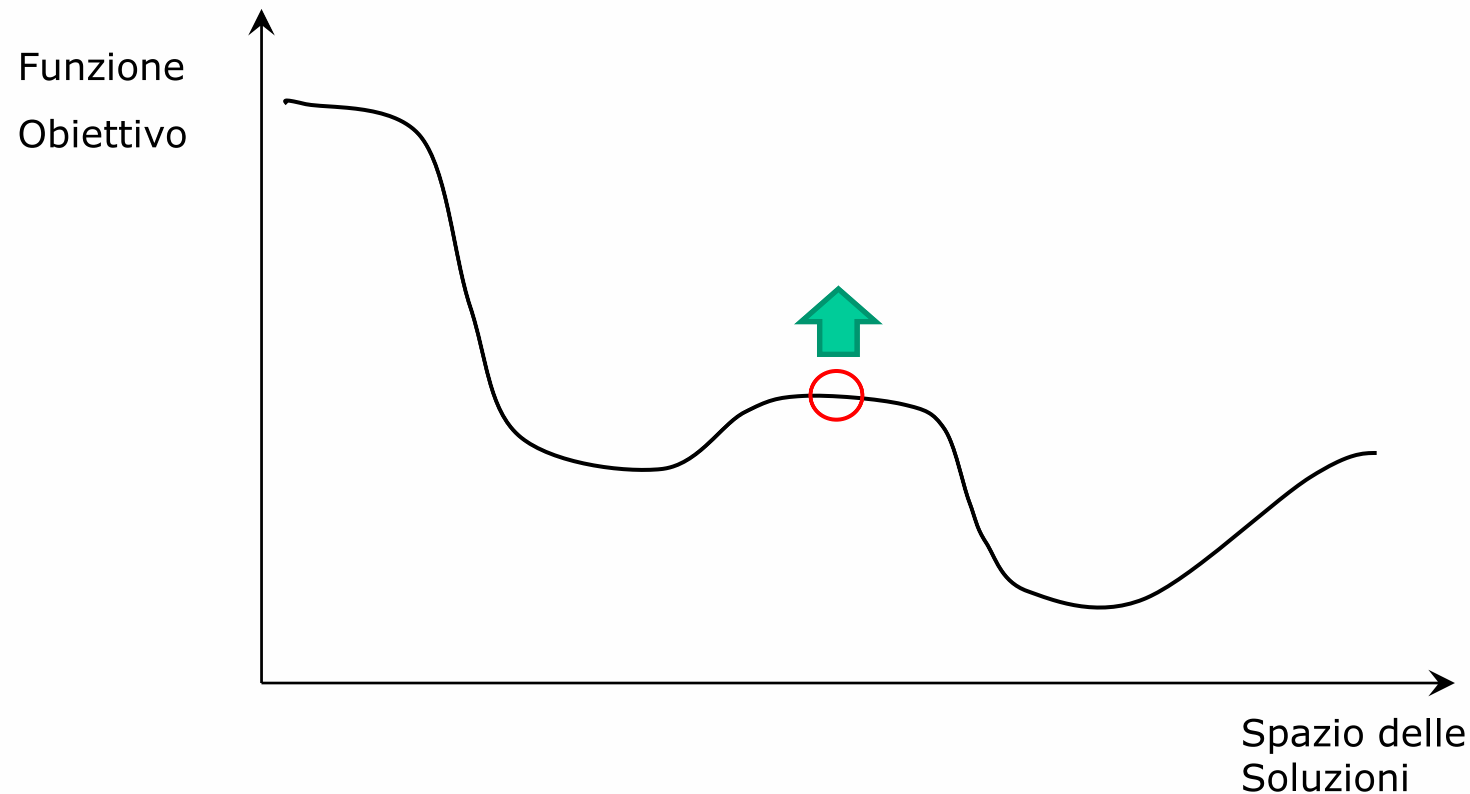
Overview (2)

- Grazie alla possibilità di poter scegliere soluzioni peggiorative, il SA è più robusto agli ottimi locali rispetto ad un hill climber



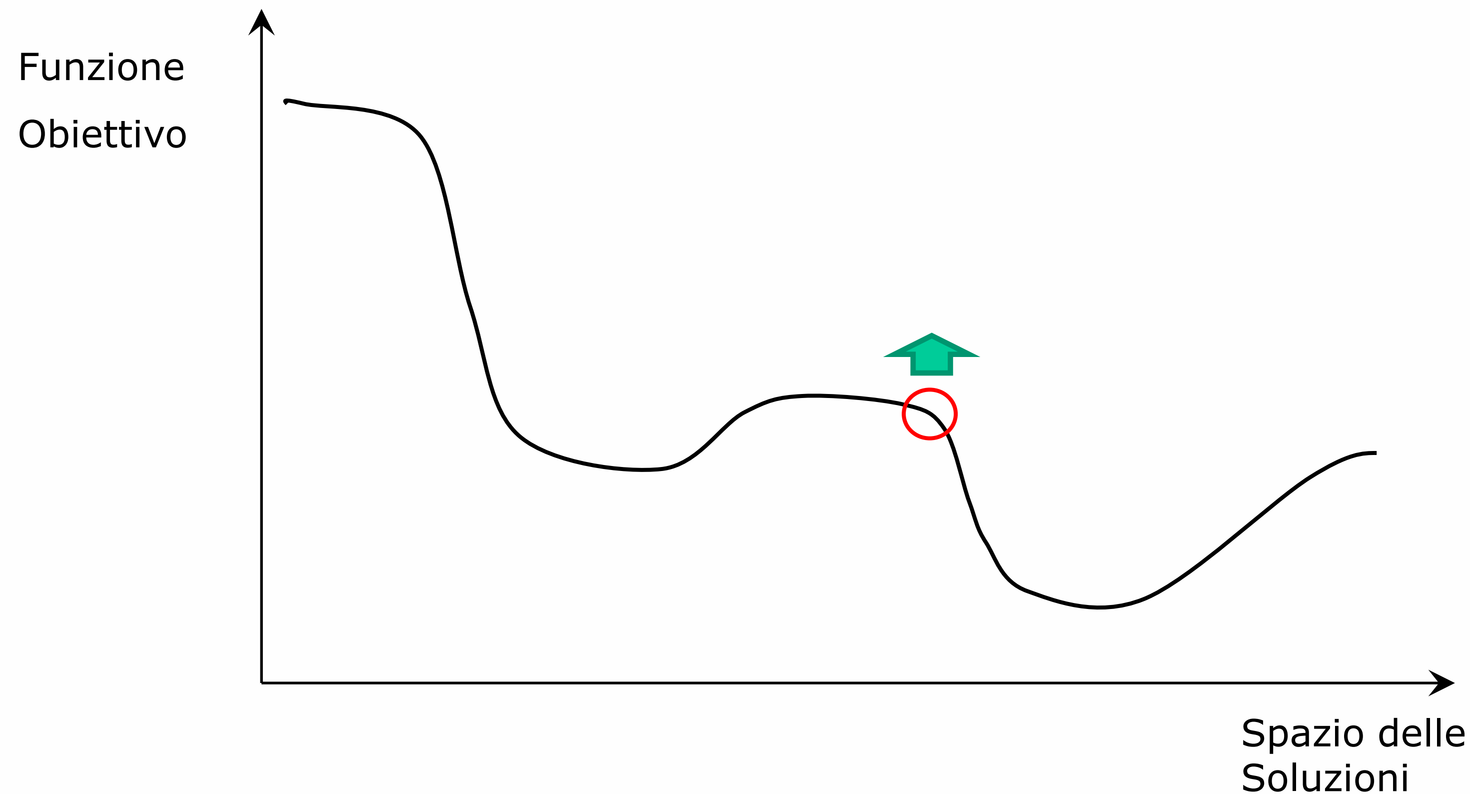
Overview (2)

- Grazie alla possibilità di poter scegliere soluzioni peggiorative, il SA è più robusto agli ottimi locali rispetto ad un hill climber



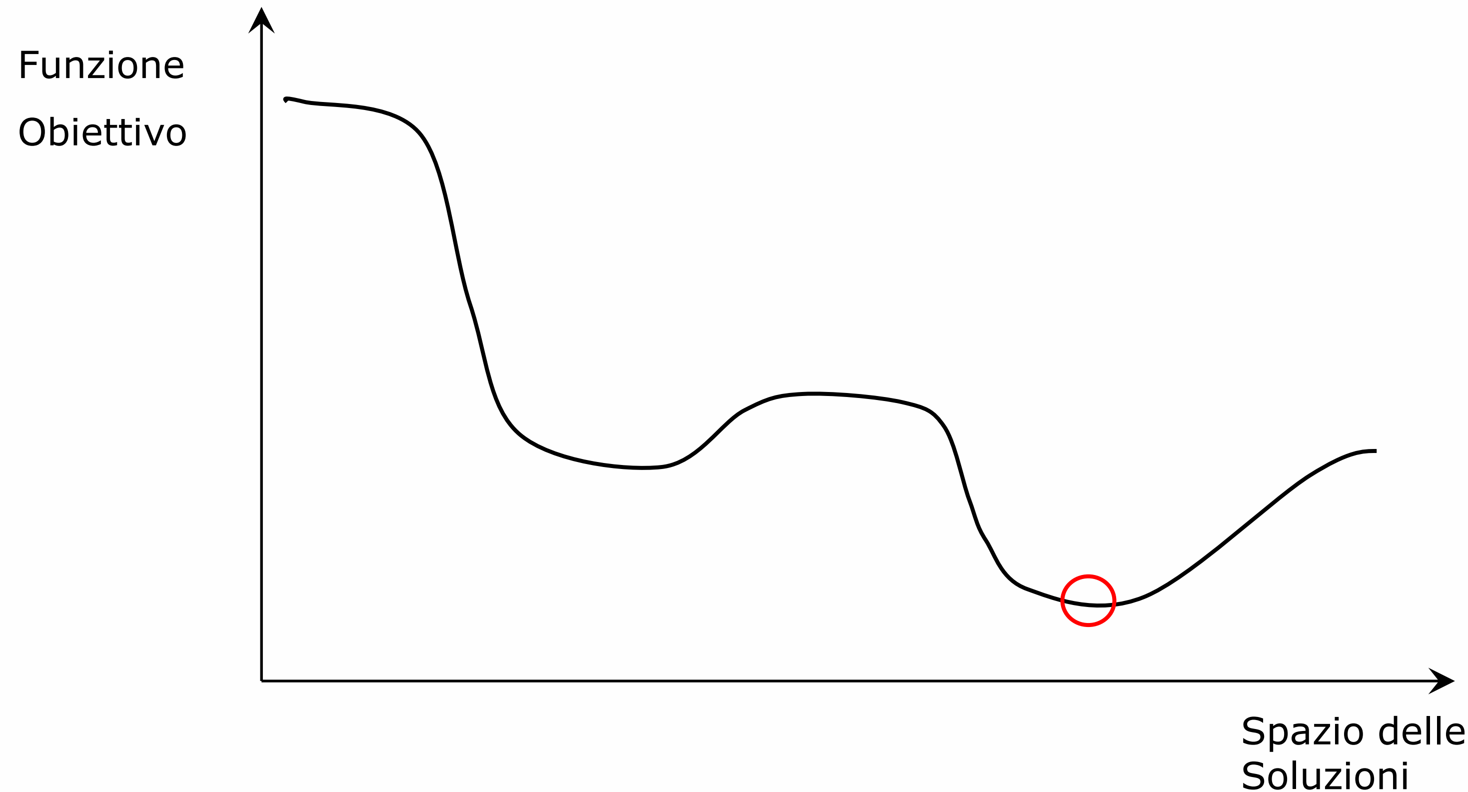
Overview (2)

- Grazie alla possibilità di poter scegliere soluzioni peggiorative, il SA è più robusto agli ottimi locali rispetto ad un hill climber



Overview (2)

- Grazie alla possibilità di poter scegliere soluzioni peggiorative, il SA è più robusto agli ottimi locali rispetto ad un hill climber



- Il design della funzione T(i) è critica per ottenere buone prestazioni e richiede conoscenze di dominio

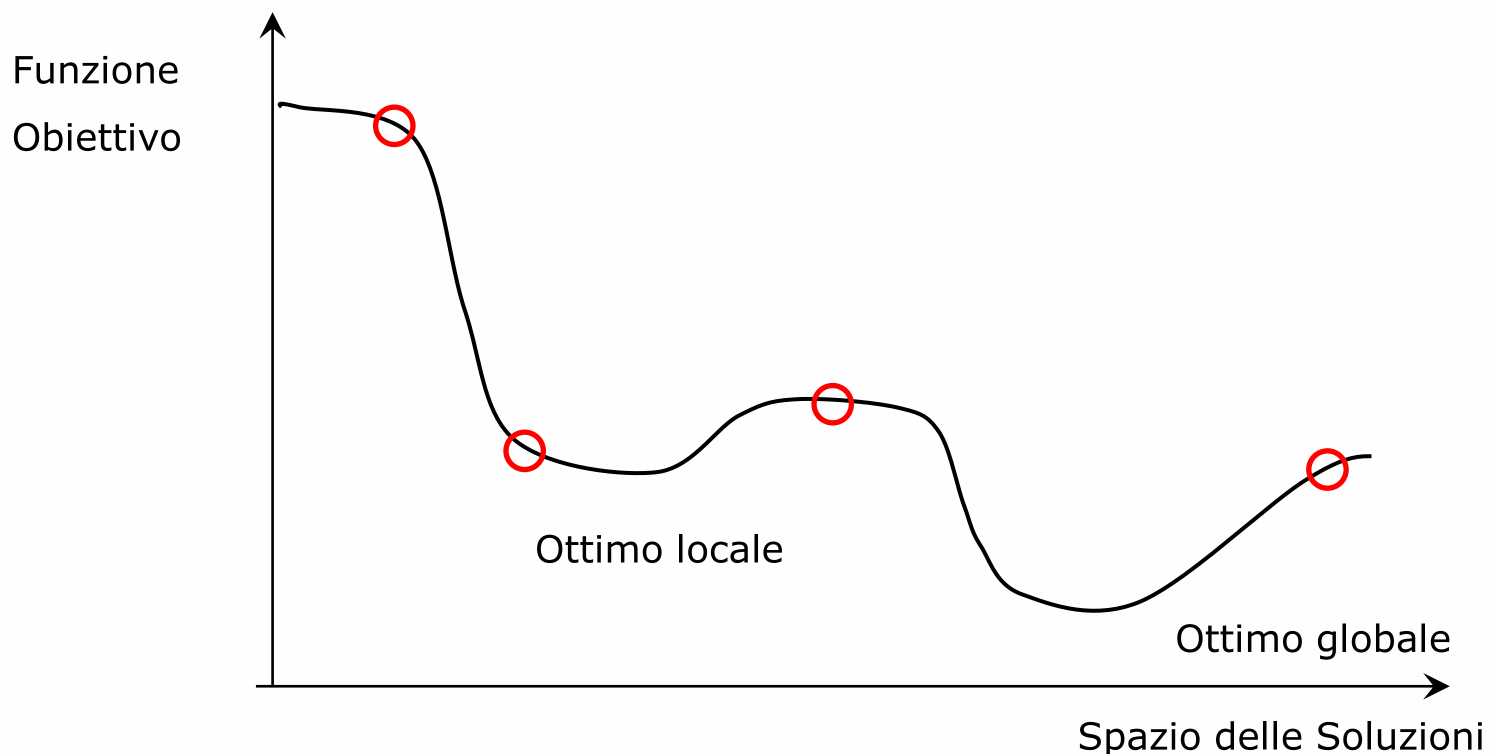
Algoritmo Base


```
genera una soluzione iniziale s di valore z(s)
i=1
while (not CRITERIO_TERMINAZIONE) {
   genera l'intorno N(s)
   trovato = false
   while not trovato {
        sceglie casualmente s' ∈ N(s)
        if (z(s') < z (s) || random() < e^((z(s)-z(s'))/T(i)) {
              s = s'
              trovato = true
        }
   }
   i=i+1
}
s*=s /*salva la migliore soluzione*/
```

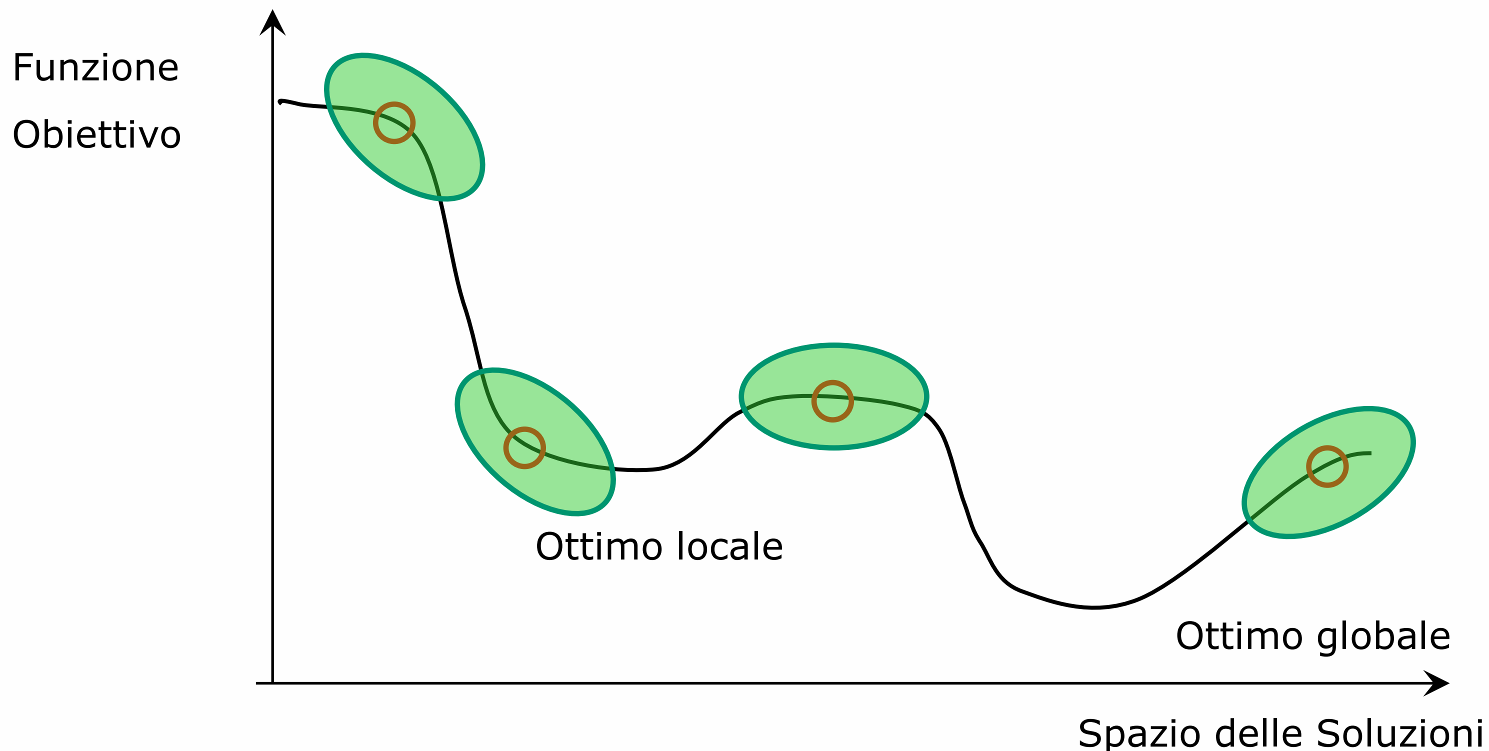
Local Beam Search

Overview

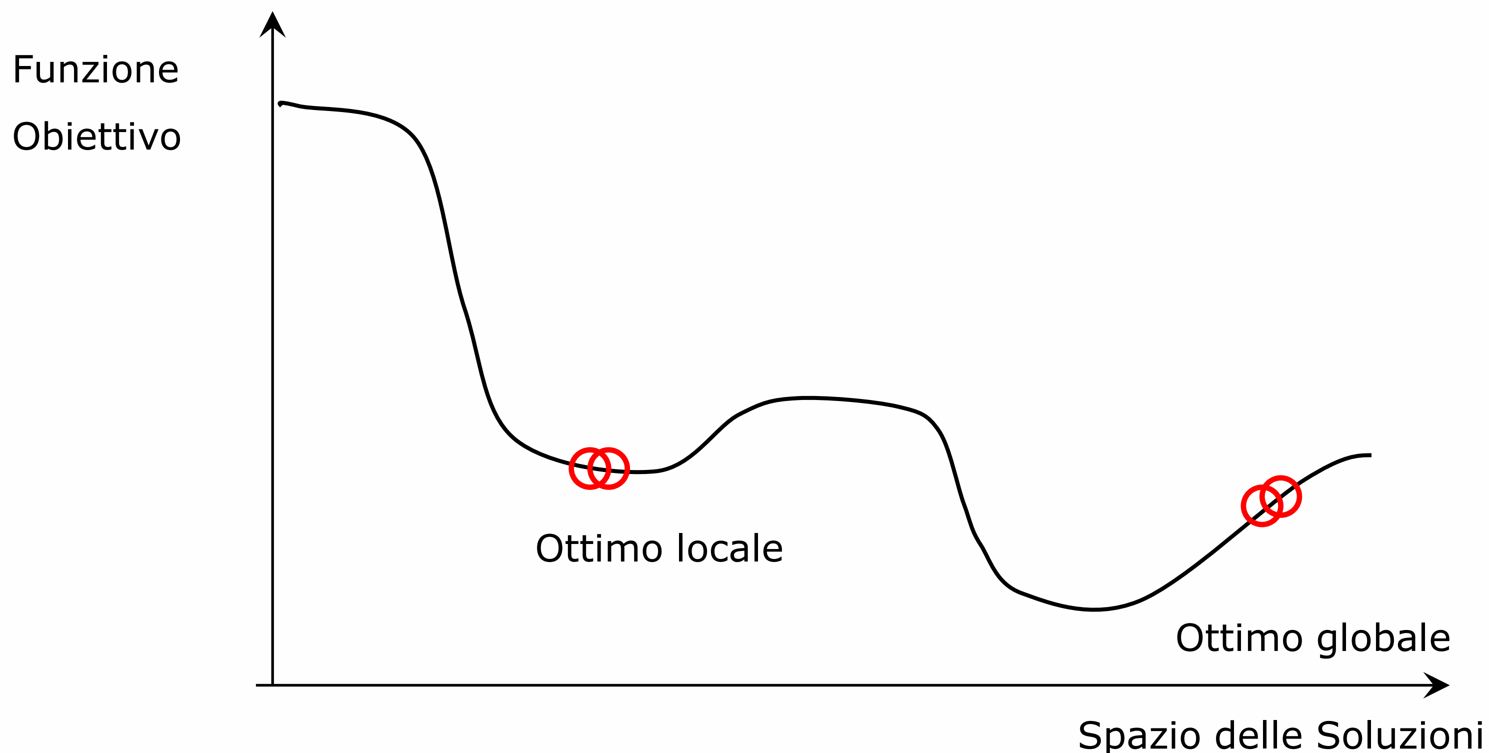
- Invece di focalizzarsi su una singola soluzione corrente, effettua la ricerca nell'intorno di **K** soluzioni diverse
  - L'algoritmo parte da **K** soluzioni scelte casualmente
  - Ad ogni iterazione genera l'intorno delle **K** soluzioni correnti
  - Sceglie come nuove soluzioni correnti le migliori **K** fra tutte quelle analizzate



- Invece di focalizzarsi su una singola soluzione corrente, effettua la ricerca nell'intorno di **K** soluzioni diverse
  - L'algoritmo parte da **K** soluzioni scelte casualmente
  - Ad ogni iterazione genera l'intorno delle **K** soluzioni correnti
  - Sceglie come nuove soluzioni correnti le migliori **K** fra tutte quelle analizzate

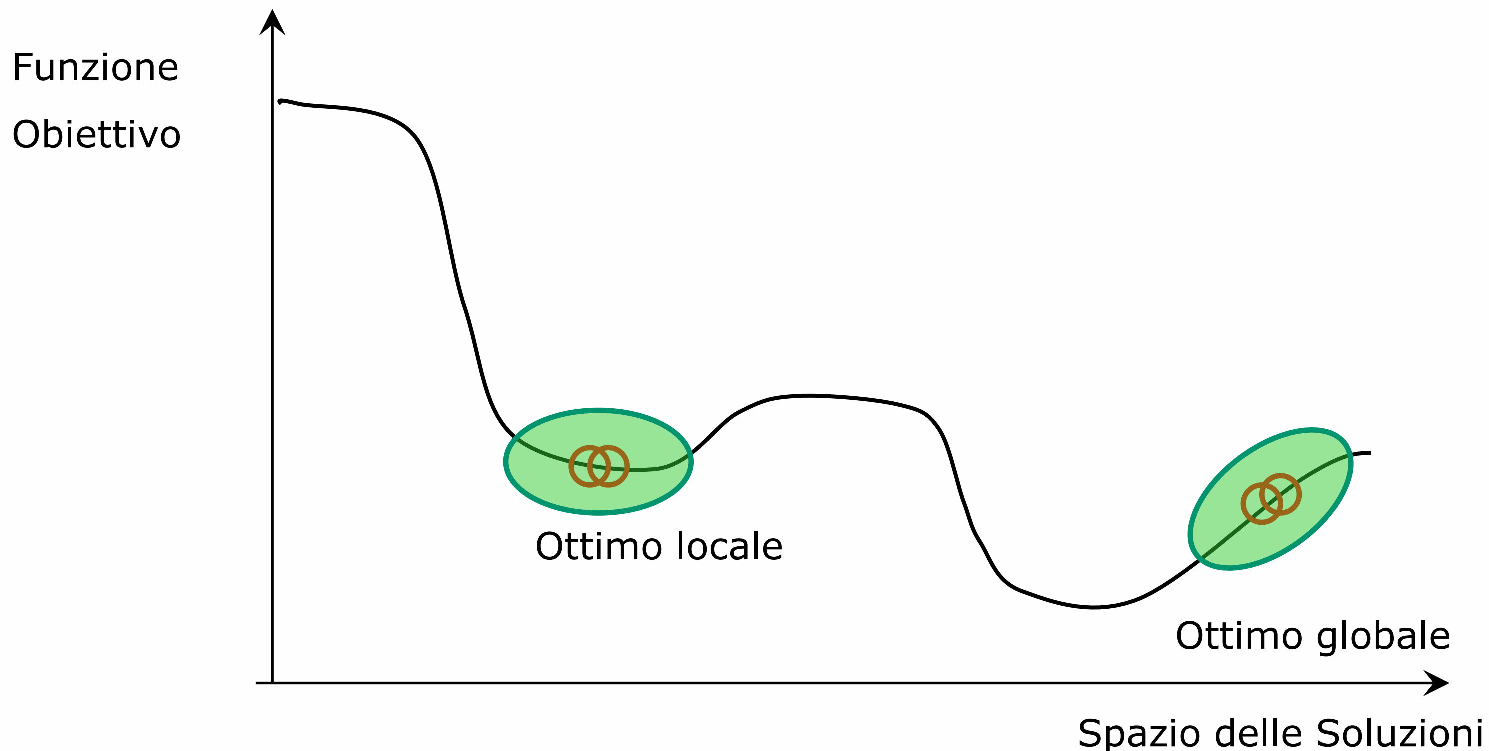


- Invece di focalizzarsi su una singola soluzione corrente, effettua la ricerca nell’intorno di K soluzioni diverse
  - L’algoritmo parte da K soluzioni scelte casualmente
  - Ad ogni iterazione genera l’intorno delle K soluzioni correnti
  - Sceglie come nuove soluzioni correnti le migliori K fra tutte quelle analizzate

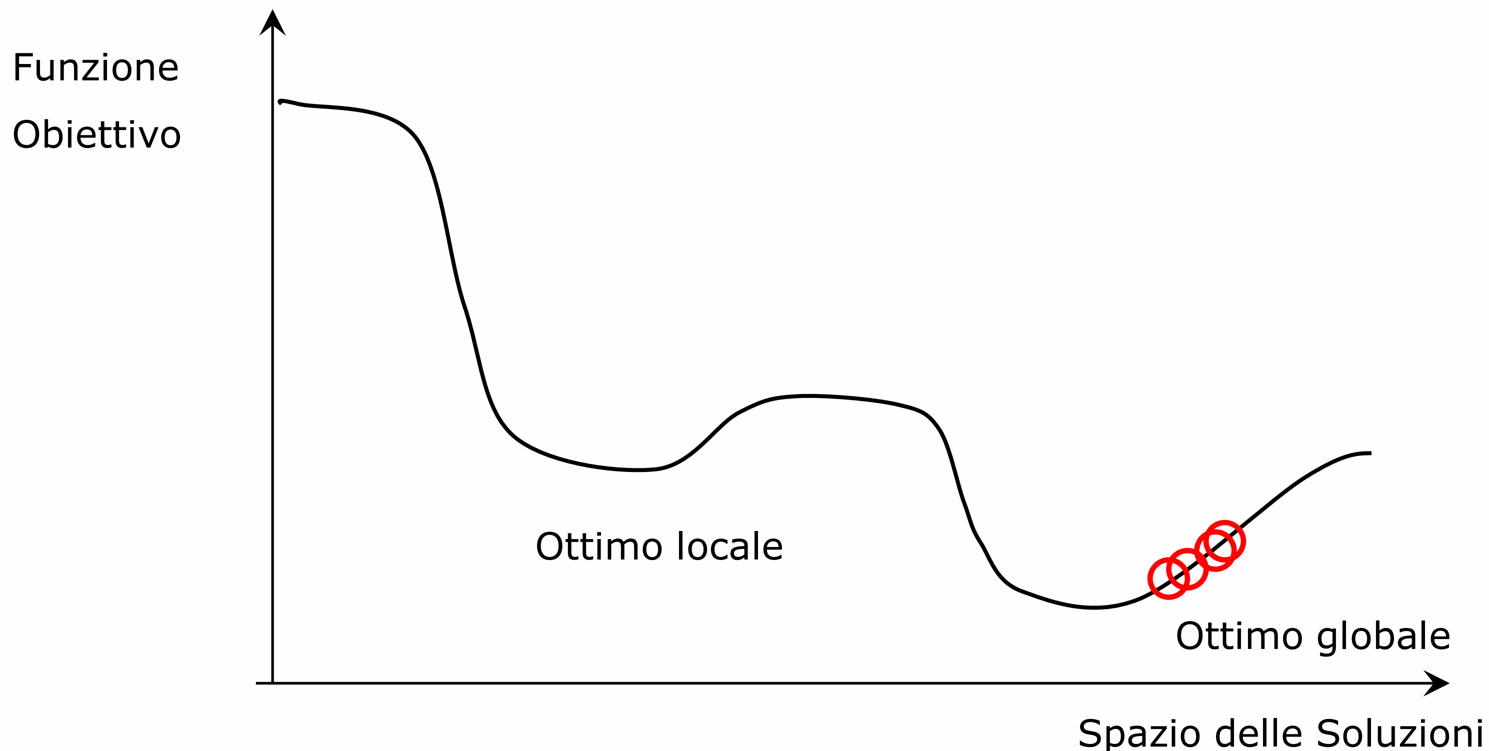


Overview

- Invece di focalizzarsi su una singola soluzione corrente, effettua la ricerca nell'intorno di **K** soluzioni diverse
  - L'algoritmo parte da **K** soluzioni scelte casualmente
  - Ad ogni iterazione genera l'intorno delle **K** soluzioni correnti
  - Sceglie come nuove soluzioni correnti le migliori **K** fra tutte quelle analizzate



- Invece di focalizzarsi su una singola soluzione corrente, effettua la ricerca nell'intorno di K soluzioni diverse
  - L'algoritmo parte da K soluzioni scelte casualmente
  - Ad ogni iterazione genera l'intorno delle K soluzioni correnti
  - Sceglie come nuove soluzioni correnti le migliori K fra tutte quelle analizzate



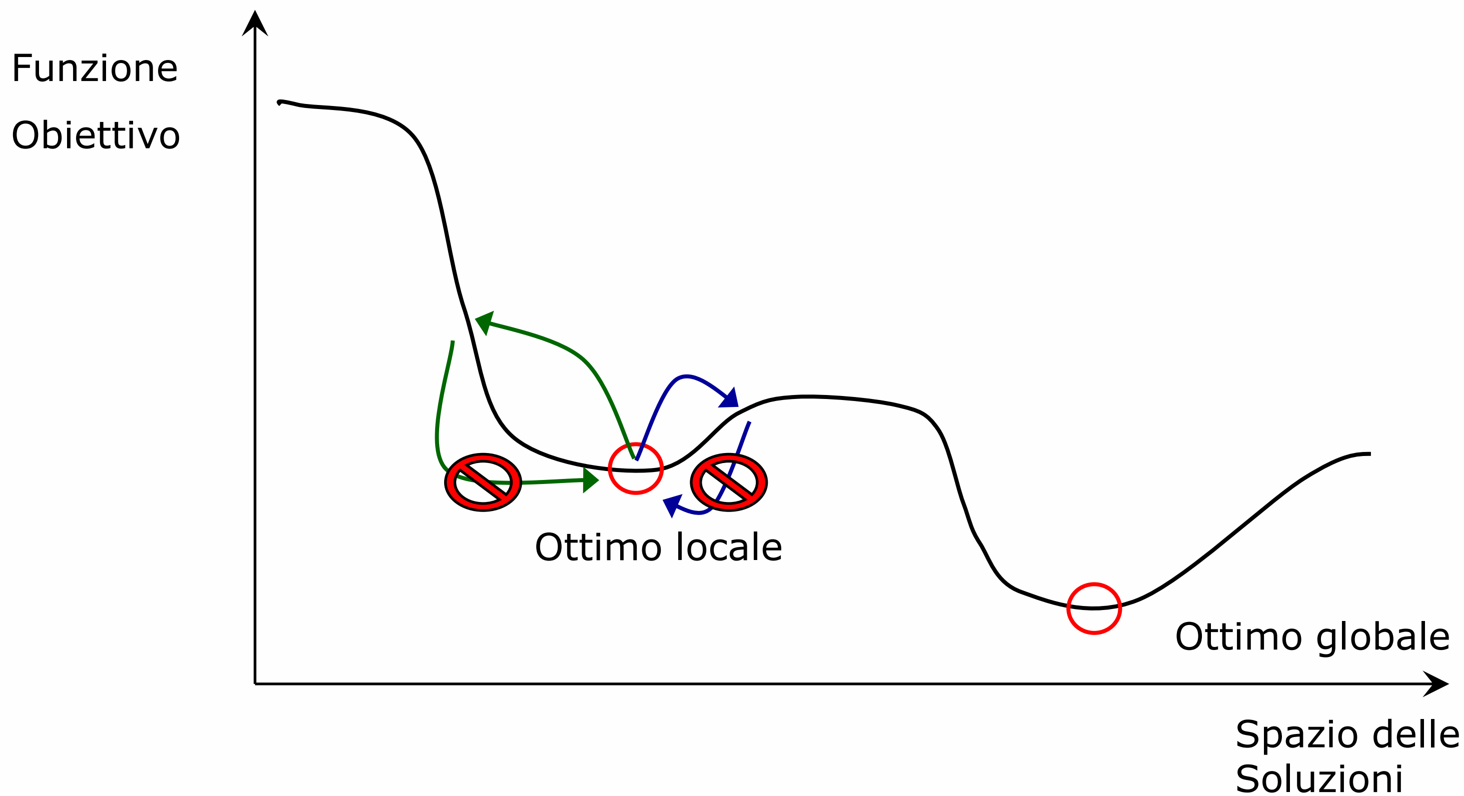
Algoritmo Base

```
genera il vettore s delle k soluzioni iniziali
while (not CRITERIO_TERMINAZIONE) {
   for (i=0; i<k; i++) {
        genera l'intorno N(s[i])
        N = N ∪ N(s[i])
    }
   trova le migliori k soluzioni s' ∈ N rispetto a z(·)
   s = s'
}
Ritorna la migliore soluzione in s rispetto a z(·)
```

Tabu Search

- È una generalizzazione della ricerca locale che consente di esplorare soluzioni "peggiorative"
- Utilizza una memoria a breve termine (**Tabu List**) per evitare di ritornare nelle ultime *t* soluzioni visitate:

$$T = \{x_{k\text{-}1}, x_{k\text{-}2}, \ldots, x_{k\text{-}t}\}$$



- Pseudocodice dell'algoritmo base per un problema di minimizzazione
  - $z(\cdot)$ è la funzione obiettivo
  - $N(s)$ è l'intorno di s

```
genera una soluzione iniziale s di valore z(s)
s* = s, k = 1, T = {s}
while (not CRITERIO_TERMINAZIONE) {
   genera l'intorno N(s)\T /* non tabu */
   trova la migliore soluzione s' ∈ N(s)\T rispetto a z(·)
   if (z(s') < z (s*)) {
        s* = s'
        kbest = k }
   s = s'
   k = k+1
   inserisci s' in T al posto della soluzione più vecchia
endwhile
```

Tabu Search: criteri di terminazione

- Criteri di arresto possibili (`CRITERIO_TERMINAZIONE`):
  - $N(s) \setminus T = \emptyset$
  - $k > k_{max}$
  - Limite di tempo raggiunto
  - $k - k_{best} > k^*$ (numero massimo di iterazioni senza miglioramento)
  - s^* ottima (ad esempio pari ad un lower bound)

Lista Tabu

- T impedisce il verificarsi di cicli di lunghezza ≤ $|T|$ ma…
- … memorizzare in T soluzioni complete può essere oneroso
- Esempio: problema di ottimizzazione in n variabili
  - ogni soluzione è un vettore di n elementi ;
  - confrontare due soluzioni costa $O(n)$;
  - verificare se una soluzione è tabu costa $O(n \cdot |T|)$
- Soluzioni
  - Memorizzare la tabu list T in una **hashtable**
  - Rappresentazione di T basata sulle mosse

- Senza perdere in generalità consideriamo un problema di ottimizzazione in cui le soluzioni sono vettori di **n** numeri interi definiti in **[a,b]**
- Memorizziamo la tabu list in una hashtable per agevolare la costruzione dell'intorno ammissibile
- Quale funzione di hash usare?

$$h(s) = \sum_{i=1}^{n} k_i s_i$$

  - Dove k è un vettore di numeri pseudocasuali definiti in [1,m]
- Caratteristiche
  - La probabilità che due vettori abbiano la stessa hash decresce con l'aumentare di n, m e b-a
  - È facile da calcolare per elementi nell'intorno: se s e s' sono identici a meno dell'i-esima componente

$$h(s') = h(s) + k_i(s'_i - s_i)$$

Mosse e Lista Tabu

- In diversi problemi di ottimizzazione, l'intorno di una soluzione e la la lista tabu possono essere definiti efficacemente con il concetto di mossa
- Una **mossa** m è un operazione elementare (e.g. scambio di due elementi di una soluzione) per ottenere una soluzione s' dalla soluzione corrente s:

$$s' = s \oplus m$$

- Possiamo allora definire l'intorno di s come:

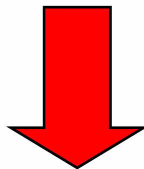
$$N(s) := \{ s' : \exists\ m \text{ tale che } s' = s \oplus m \}$$

- La tabu list può essere definita come l'insieme di **mosse inverse** alle ultime t mosse effettuate nell'esplorazione dello spazio delle soluzioni

Mosse e Lista Tabu: esempio

- Siano le soluzioni di un problema tutte le possibili terne di elementi distinti dell'insieme {a,b,c,d,e}

Soluzione	Mossa	Tabu List
abc	c → d	d → c
abd	b → c	d → c c → b
acd	d → b	d → c c → b b → d



acb=abc

Limitazione delle mosse inverse

- La tabu list T^I che rappresenta le mosse inverse è molto più restrittiva di T che rappresenta le soluzioni precedenti

$$R := \{s' : \exists\, m \in T^I \text{ tale che } s' = s \oplus m\}$$

- Risulta che

$$|\,N(s) \setminus R\,| \leq |\,N(s) \setminus T\,| \text{ (spesso <<)}$$

- Inoltre T^I non garantisce che non si abbiano cicli di periodo $\leq |\,T^I\,|$

Intensificazione e diversificazione

- La tabu list costituisce la memoria di breve periodo
- Una ricerca efficace necessita anche di memoria di medio/lungo periodo
- Il Tabu Search utilizza a questo scopo i seguenti meccanismi
  - **intensificazione della ricerca**
    - non è consentito spostarsi troppo dalla parte di regione che si sta visitando
    - preferire mosse con caratteristiche in comune con una buona soluzione recentemente incontrata
    - penalità da aggiungere a $z(\cdot)$ per le mosse che alterano tale caratteristica
  - **diversificazione della ricerca**
    - per spostarsi verso altre zone (inesplorate) dello spazio delle soluzioni
    - penalità da aggiungere a $z(\cdot)$ per le soluzioni troppo vicine a quella corrente

Tabu search: vincoli

- Se il problema è fortemente vincolato la cardinalità di $N(s)$ può essere molto piccola (è facile che $N(s) \setminus T = \emptyset$)
- Si rilassano alcuni vincoli aggiungendo a $z(\cdot)$ una penalità proporzionale alla violazione dei vincoli in s
- La ricerca si muove anche attraverso soluzioni non ammissibili
- Aggiustamento adattativo della penalità:
  - La penalità può crescere se da molte iterazioni non si incontrano soluzioni ammissibili
  - La penalità può decrescere se da molte iterazioni si incontrano soluzioni tutte ammissibili

Tabu Tenure

- Finora abbiamo ipotizzato la **tabu tenure** (lunghezza della lista tabu) una costante pari a t
- Nella pratica può essere modificata dinamicamente (es. ogni h iterazioni)
  - **Intensificazione:**
    se s^* è stata migliorata → $t := \min\{t_{min}, t-1\}$
  - **Diversificazione:**
    se s^* rimane immutata → $t := \max\{t_{max}, t+1\}$
- In alternativa può essere scelta casualmente in $[t_{min}, t_{max}]$
- È molto importante scegliere I valori di t, t_{min}, t_{max}, h