

Intelligenza Artificiale

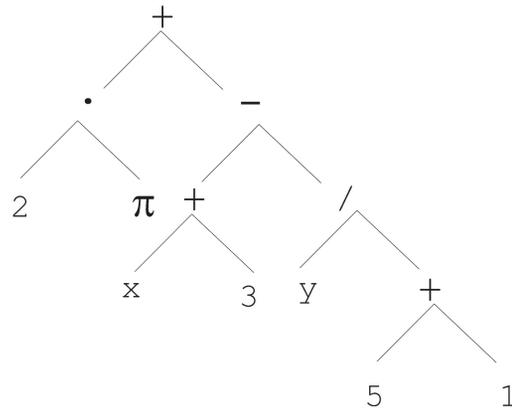
Genetic Programming

Marco Piastra

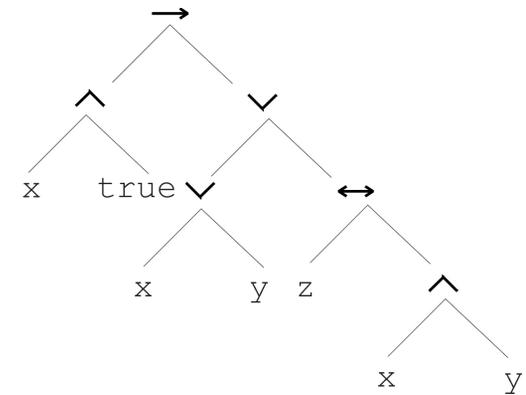
Strutture sintattiche e alberi

- Qualsiasi cosa abbia una sintassi formale (una grammatica) può essere rappresentata in forma di albero

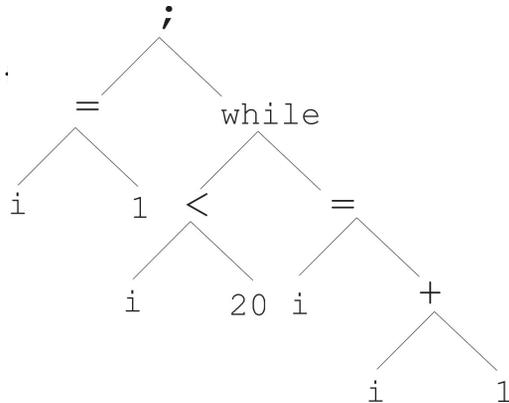
$$2 \cdot \pi + \left((x+3) - \frac{y}{5+1} \right)$$



$$(x \wedge z) \rightarrow ((x \vee y) \vee (z \leftrightarrow (x \wedge y)))$$



```
i = 1;
while (i < 20)
    i = i + 1
}
```



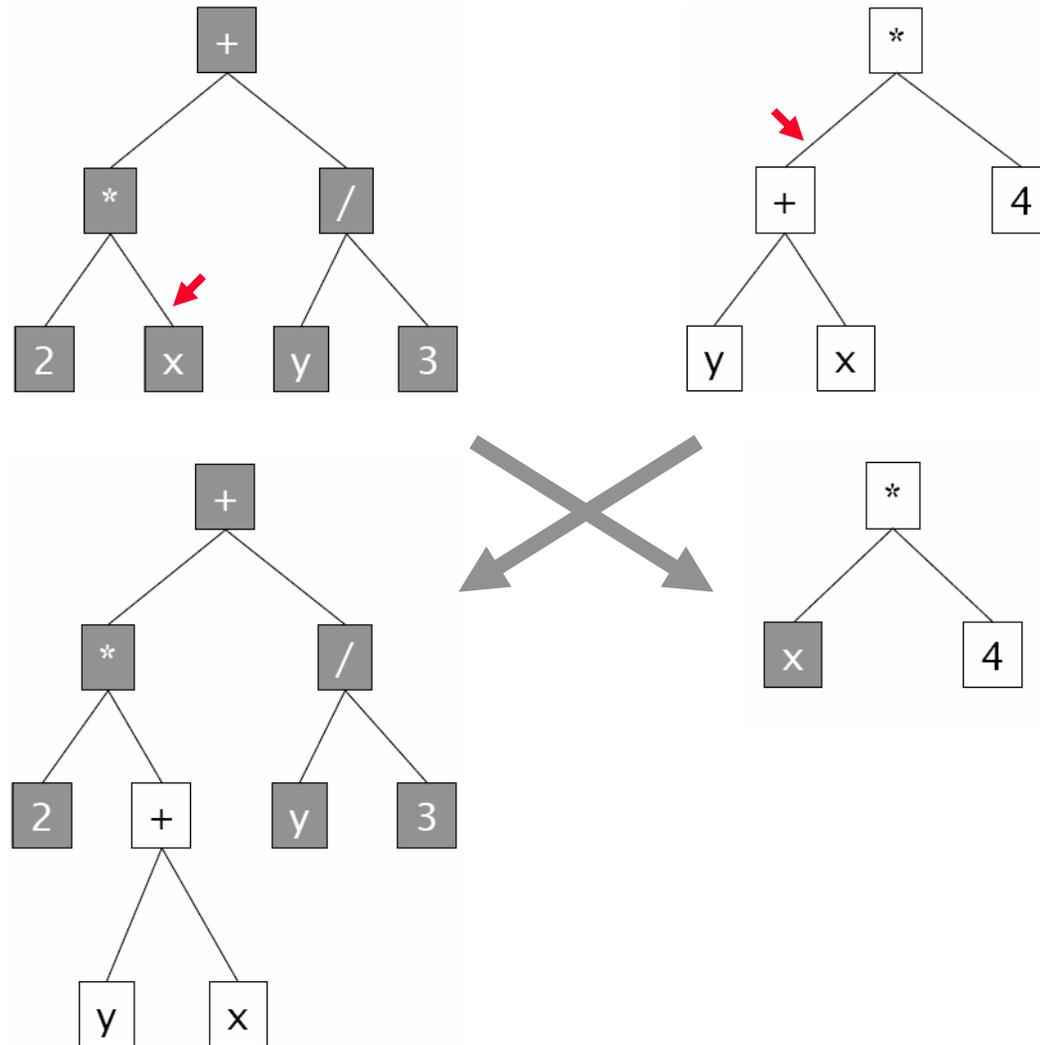
Genetic Programming

- Una tecnica di calcolo evolutivo per le **strutture ad albero**
 - Le strutture ad albero rappresentano *espressioni* in una *sintassi*
 - Gli operatori genetici (*mutazione, ricombinazione*) agiscono direttamente sulle strutture ad albero
- Valutazione della *fitness*
 - Le strutture ad albero hanno una *semantica operativa*
 - Descrivono operazioni da compiere
 - p.es. il programma di controllo di un robot
 - La valutazione della *fitness* è indiretta
 - Si valuta la struttura ad albero
 - La funzione di fitness si applica ai risultati della valutazione
- Lo spazio delle possibili soluzioni
 - p.es. lo spazio delle espressioni aritmetiche o lo spazio dei programmi in un linguaggio di programmazione

Operatori di ricombinazione

- **Crossover**

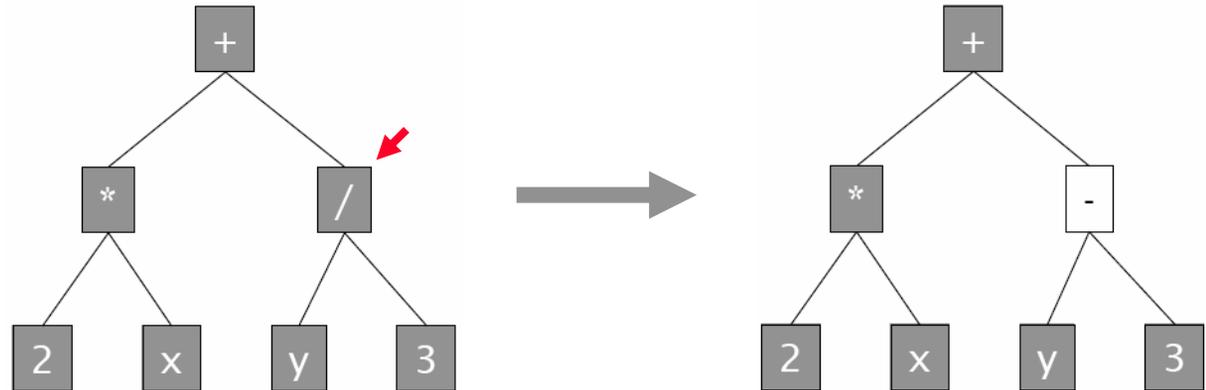
- Selezione di due punti di taglio
- Scambio dei sotto-alberi



Operatori di mutazione

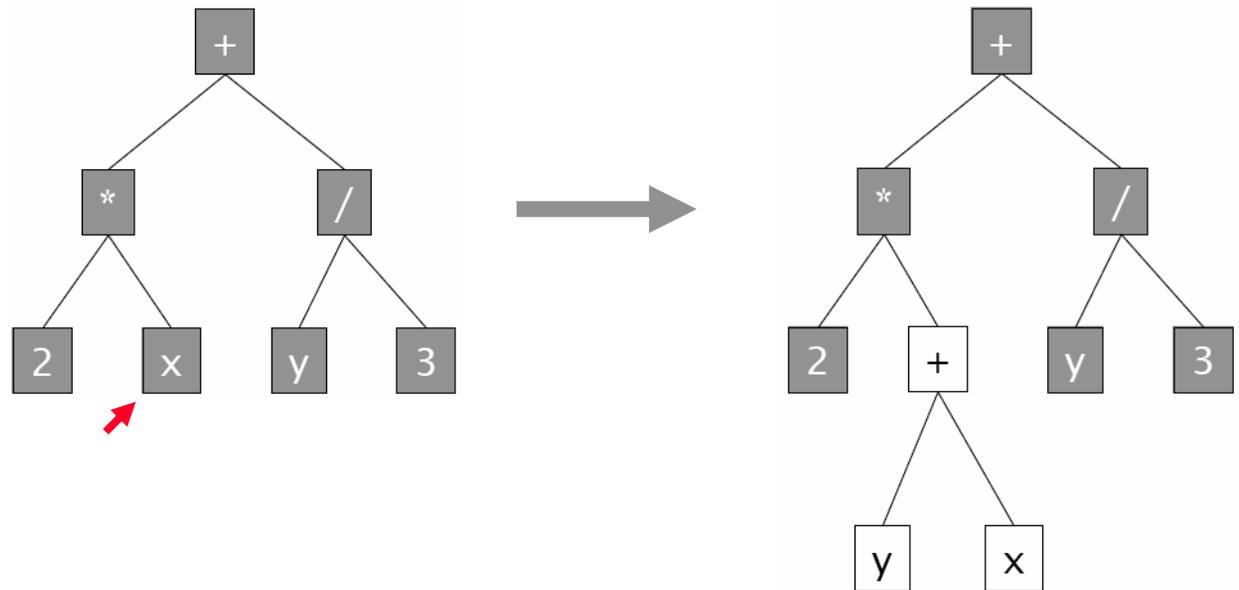
- **Sostituzione**

- Selezione di un nodo
- Alterazione del nodo



- **Editing**

- Selezione di un nodo
- Rimozione del sotto-albero
- Generazione di un nuovo sotto-albero



Generazione di strutture ad albero

- Algoritmo di generazione
 - Due insiemi di simboli:
 - Simboli non terminali NT, con *arietà* $\{+, -(2), *, /, -(1)\}$
 - Simboli terminali T $\{x, y, z, \pi, \dots\}$
 - Limite di profondità massima D_{\max}
 - Si genera l'albero in modo incrementale
 - A partire dal nodo radice, scegliendo i nodi a caso in NT e T
- *Full method*
 - Ogni nodo a profondità $< D_{\max}$ è scelto in NT
 - Ogni nodo a profondità $= D_{\max}$ è scelto in T
- *Grow Method*
 - Ogni nodo a profondità $< D_{\max}$ è scelto in $NT \cup T$
 - Ogni nodo a profondità $= D_{\max}$ è scelto in T
 - Gli alberi risultanti possono avere profondità $< D_{\max}$

Strategie di generazione

- Popolazione iniziale
 - Strategia classica nel GP: *ramped half-and-half*
 - 50% di individui generati con il *Full Method*
 - 50% di individui generati con il *Grow Method*
- Sintassi semplici e limitazioni semantiche
 - L'algoritmo di generazione funziona solo per le sintassi **typeless**
 - A parte i limiti di profondità
 - Qualsiasi elemento di $NT \cup T$ può essere scelto ad ogni passo dell'algoritmo di generazione
 - Le operazioni devono essere protette (p.es. divisione per zero)
 - La maggior parte dei linguaggi di programmazione sono invece **strongly-typed** (p.es. C, C++, Java, SQL)

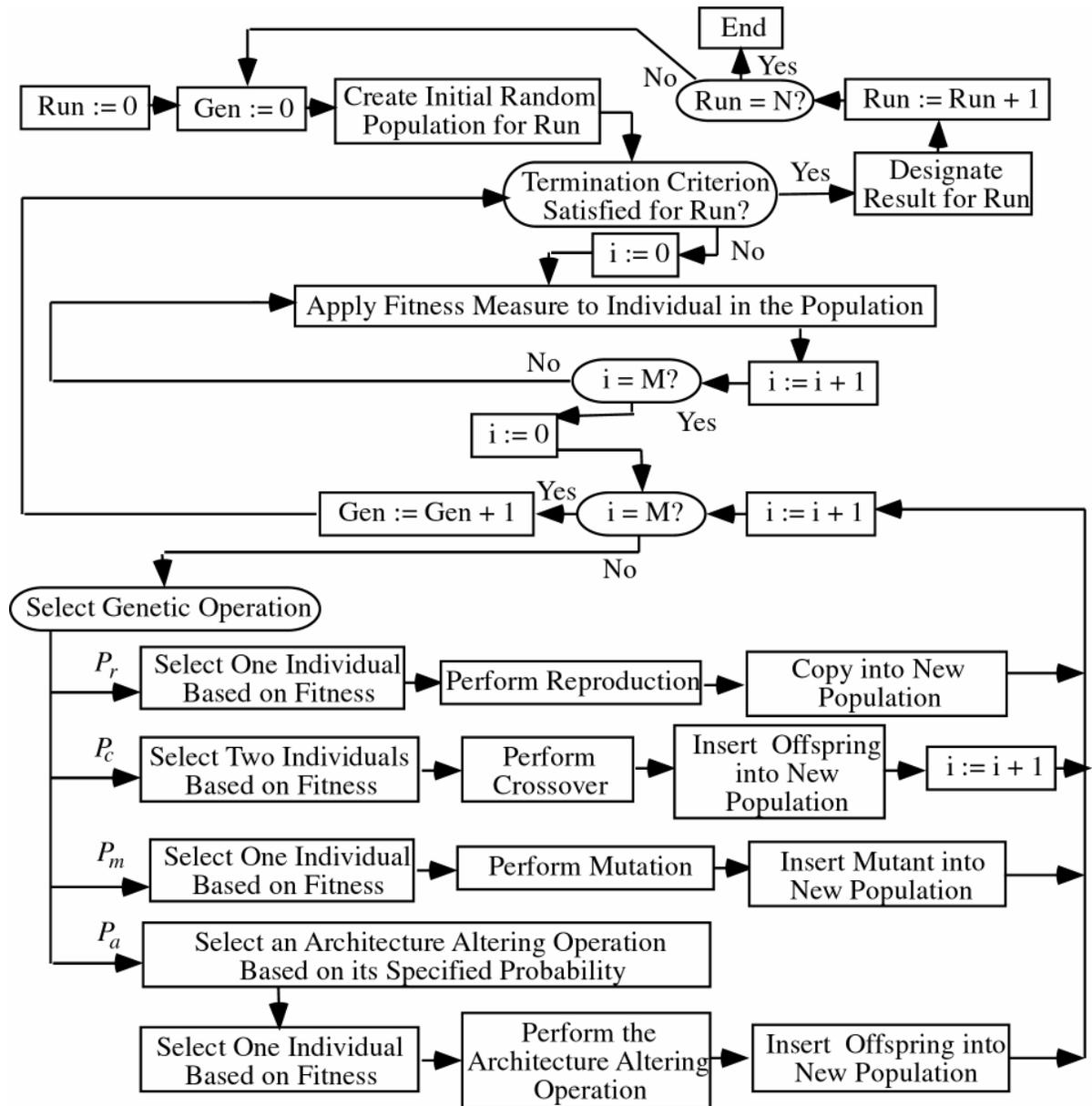
Selezione a torneo (*tournament*)

- Motivazione
 - I metodi *fitness proportionate* richiedono strutture dati di popolazione
 - Può essere scomodo su popolazioni vaste e/o distribuite
- Metodo generale
 - Si scelgono a caso k individui della popolazione
 - Tra i k individui, si sceglie quello a *fitness* più alta
- Controllo
 - Tramite il valore di k (finestra di selezione)
 - Per $k = 1$ la selezione è puramente casuale (la *fitness* non conta più)
 - Per $k = \text{dim}(\text{popolazione})$ la selezione non è più casuale
 - Maggiore il valore di k , maggiore è la pressione selettiva
 - Gli individui a minore fitness hanno probabilità minori di essere selezionati
 - Maggiore è la pressione selettiva, più breve è la durata della diversità
 - L'individuo migliore (della popolazione) prende il sopravvento
 - Un alto tasso di mutazione non può compensare questo effetto

Riproduzione ed elitismo

- Motivazione
 - Nei processi evolutivi generazionali, è pericoloso produrre nuove generazioni solo per mutazione e ricombinazione
 - Si rischia di perdere i risultati acquisiti
 - In quanto gli individui migliori potrebbero andar persi
- **Riproduzione**
 - Gli individui selezionati vengono copiati nella nuova generazione
 - Un operatore genetico 'degenere': non fa nulla
- **Elitismo**
 - Selezione non casuale dei migliori n individui e riproduzione
 - Si ha la certezza di non perdersi i migliori di ogni generazione
 - Usare con cautela: abbrevia la durata della diversità

Schema di flusso



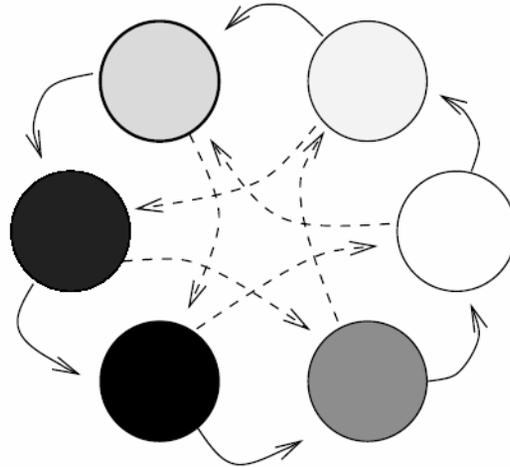
Parametri di processo: esempio

- Popolazione
 - Dimensione: da qualche decina a decine di migliaia
 - Dominante, in molti casi, è il costo di valutazione della *fitness*
 - Popolazioni molto ampie danno maggiori garanzie ma rallentano il processo
 - Profondità D_{max} : tipicamente 7 o 8 (dipende dal problema)
- Selezione operatori genetici
 - Riproduzione: $p = 0.1$
 - Crossover: $p = 0.7$
 - Mutazione: $p = 0.2$
 - La mutazione per *editing* è meno distruttiva
 - Elitismo: poche unità
- Pressione selettiva
 - Finestra di selezione k : tipicamente 5 su una popolazione di 1024

Evoluzione *steady-state*

- Principio di base
 - In un processo generazionale
 - Si hanno successive popolazioni (generazioni)
 - Prodotte iterativamente per selezione ed applicazione degli operatori genetici
 - In un processo **steady-state**
 - Si ha un'unica popolazione di dimensione costante
 - L'offspring prodotto ad ogni iterazione rimpiazza individui esistenti
 - Si usa una selezione 'inversa' per identificare gli individui da sostituire
 - Gli individui a fitness minore hanno maggiore probabilità di essere rimpiazzati
- Motivazione
 - Risparmio di memoria
 - Popolazione condivisa in un sistema di valutazione parallela
 - Diverse cpu eseguono il ciclo iterativo sulla stessa popolazione (database)

Evoluzione parallela: *island model*



- **Island model**

- Diverse popolazioni (*island*)
 - Ciascuna (tipicamente) gestita da una singola CPU
 - Gli individui sono dello stesso tipo (con qualche possibile, limitata eccezione)
- L'evoluzione avviene in modo indipendente su ciascuna *island*
- Occasionalmente, si possono avere migrazioni tra *island*
 - Selezione degli individui
 - Comunicazione via code (asincrona)

Metodi di ottimizzazione a confronto

- Metodi **deterministici**

- Iterazioni successive con passi determinati in modo deterministico
- Tipicamente si basano su proprietà della funzione da ottimizzare
 - i.e. derivabilità
- Esempi:
 - semplice, steepest descent (o hill climbing), gradiente coniugato, etc.

- Metodi **stocastici**

- Iterazioni successive con passi determinati da processi aleatori
- Tipicamente non si basano su proprietà della funzione da ottimizzare
 - si dicono anche metodi 'black-box'
- Esempi:
 - calcolo evolutivo, simulated annealing, stochastic gradient, random search

No Free Lunch Theorem (Walpert & Macready, 1996)

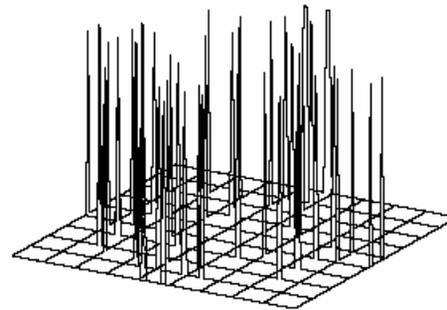
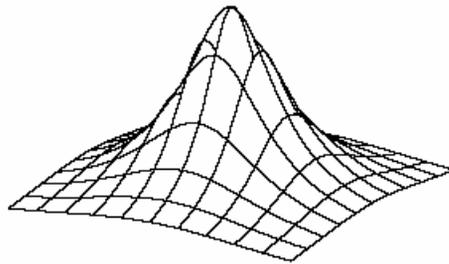
- Le performance dei metodi 'black-box'
 - In generale (su tutte le possibili funzioni da ottimizzare) sono identiche

$$\sum_f P(d_m^y | f, m, a_1) = \sum_f P(d_m^y | f, m, a_2).$$

- $P(d_m^y | f, m, a_1)$ è la probabilità che un algoritmo a_1
 - trovi un campione per cui la funzione f ha valore d_m^y
 - in esattamente m passi
 - la sommatoria è estesa a tutte le possibili funzioni f
- Significato
 - Non esiste un metodo 'black-box' migliore in tutti i casi
 - La bontà del metodo è tipicamente legata alla classe di problemi

Spazio di ricerca del GP

- Piccole variazioni del genotipo (p.es. cambio di un'istruzione) provocano grandi variazioni del valore di fitness
- Si parla di 'ruggedness' (asprezza) del *fitness landscape* in un problema GP



- Osservazioni

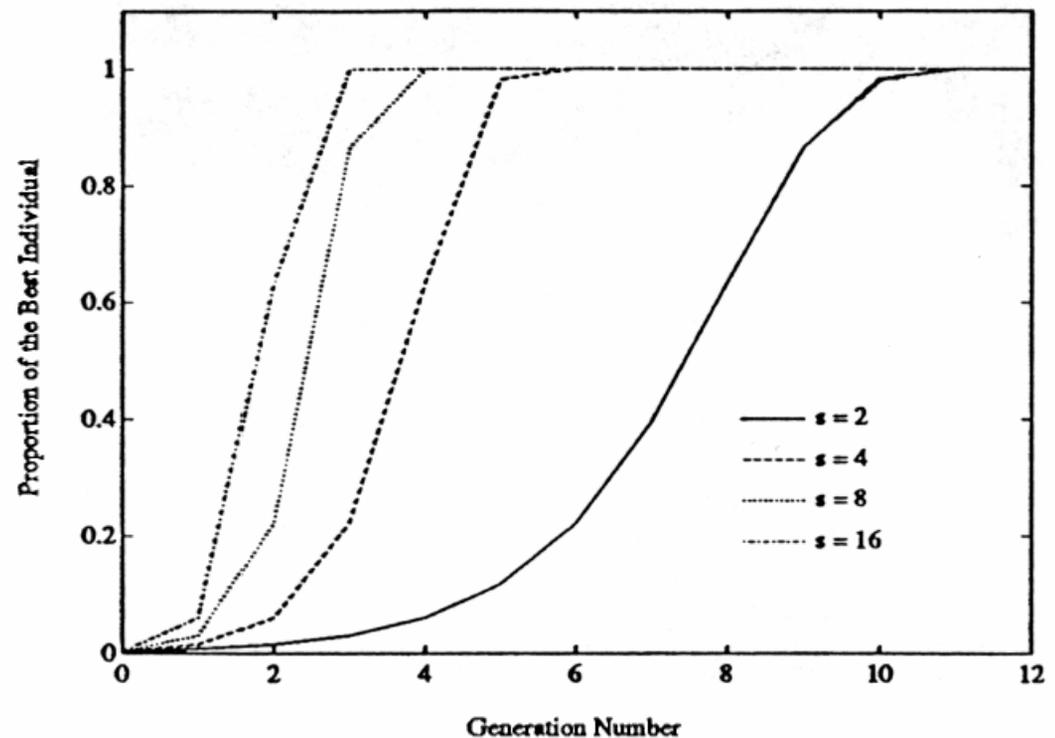
- In un simile spazio di ricerca, si possono applicare solo metodi 'black box'
- Per il 'No Free Lunch Theorem', il calcolo evolutivo (ed il GP in particolare)
 - è migliore della ricerca casuale solo in una determinata classe di problemi
 - nella pratica, spesso lo è
 - non esiste un soddisfacente inquadramento teorico della classe di problemi

Building-block hypothesis

- Dallo Schema Theorem
 - I blocchi compatti e più corti hanno maggiore probabilità di sopravvivere intatti al campionamento (crossover)
 - Aumentando quindi rapidamente la propria presenza nella popolazione
- **Building-block hypothesis**
 - Il calcolo evolutivo è particolarmente efficiente su problemi la cui soluzione è decomponibile in blocchi (di qualsiasi natura)
 - Ciò spiega la grande dipendenza dal metodo di codifica del genotipo
 - Riscontrabile nella pratica
 - Intuitivamente vero nel caso del GP (programmazione a blocchi)
 - L'ipotesi si scontra con numerosi esempi controfattuali
 - Non vale quindi in generale

Problemi tipici: *takeover*

- Perdita della diversità della popolazione
 - Un aumento eccessivo della pressione selettiva per accelerare il processo
 - p.es. aumento della dimensione della dimensione k nella selezione a *tournament*
 - Causa il prevalere di un singolo individuo
 - la popolazione perde diversità
 - il processo evolutivo si arresta

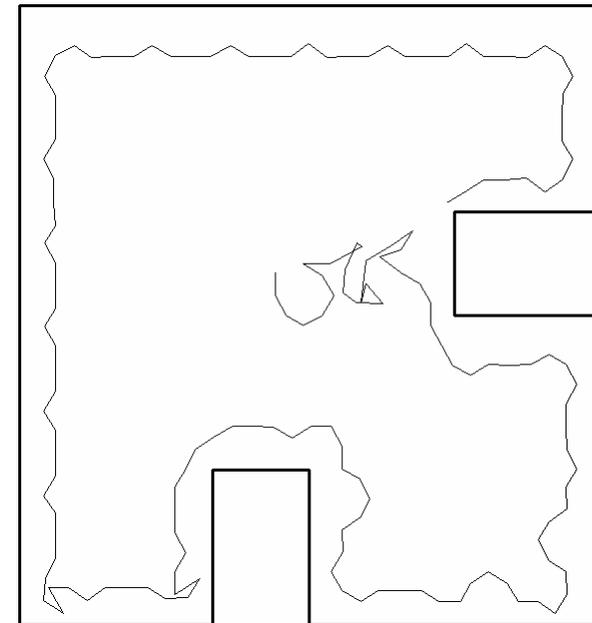
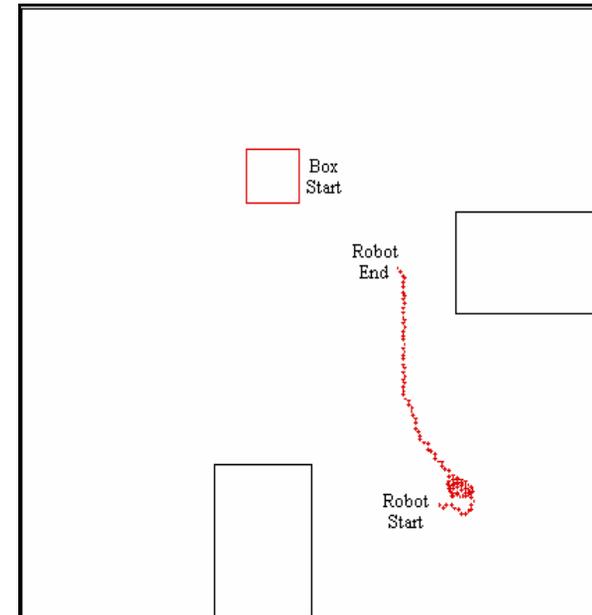


Problemi tipici: *bloating*

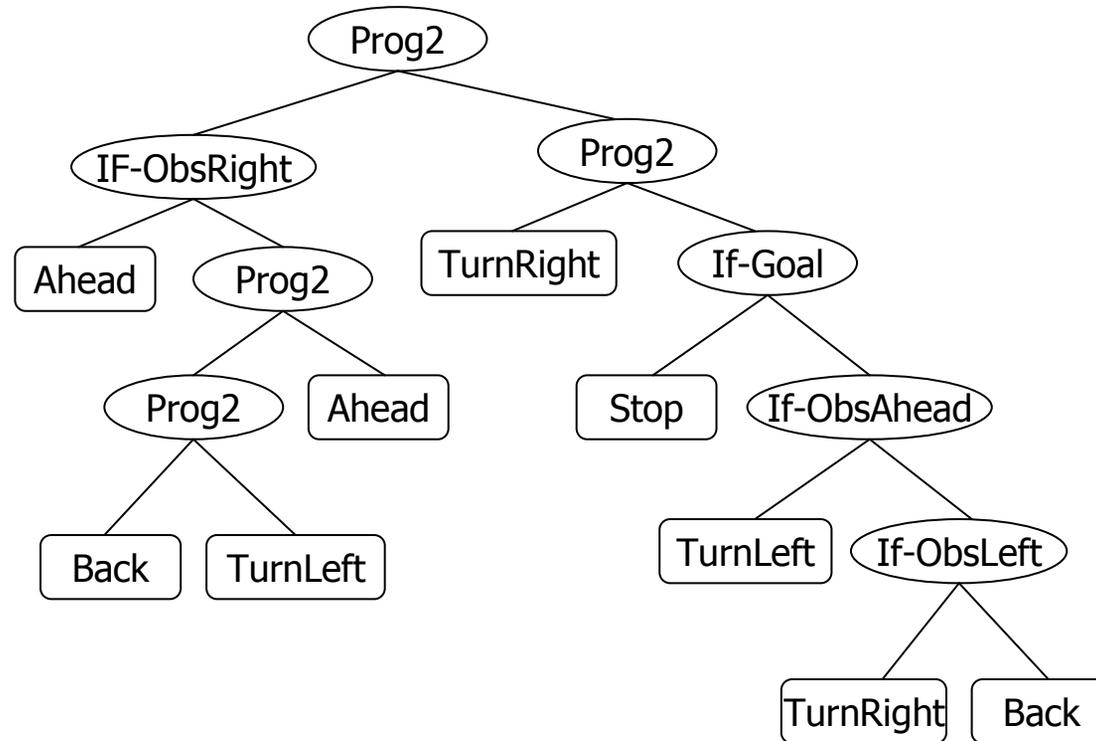
- Gigantismo degli individui (*bloating*)
 - Gli operatori genetici (p.es. *crossover*) non limitano le dimensioni dell'*offspring*
 - Individui più grandi (con parti ridondanti) preservano meglio la fitness
 - E` più alta la probabilità che gli operatori genetici alterino le parti ridondanti
 - Risultato: esplosione delle dimensioni degli individui
 - Fenomeno molto più marcato nel caso di processi non convergenti (può essere un indizio ...)
 - Rimedi:
 - In un processo convergente, spesso il problema si risolve da solo
 - Uso della *parsimony pressure*: la fitness penalizza gradualmente gli individui più grandi

Esempio: robot

- Un robot che si muove in un ambiente
 - L'obiettivo è evitare gli ostacoli
 - La logica di controllo utilizza un ciclo *sense-eval-act*:
 - Lettura dei sensori (*sense*)
 - Valuta le informazioni (*eval*)
 - Comandi agli attuatori (*act*)
 - L'obiettivo dell'evoluzione è una logica di controllo
 - Esplorazione efficace e completa
 - Senza urtare gli ostacoli
 - La *fitness* valuta il comportamento
 - + Area esplorata, – Collisioni
 - La valutazione della fitness prende tempo ...



Robot: tipico individuo



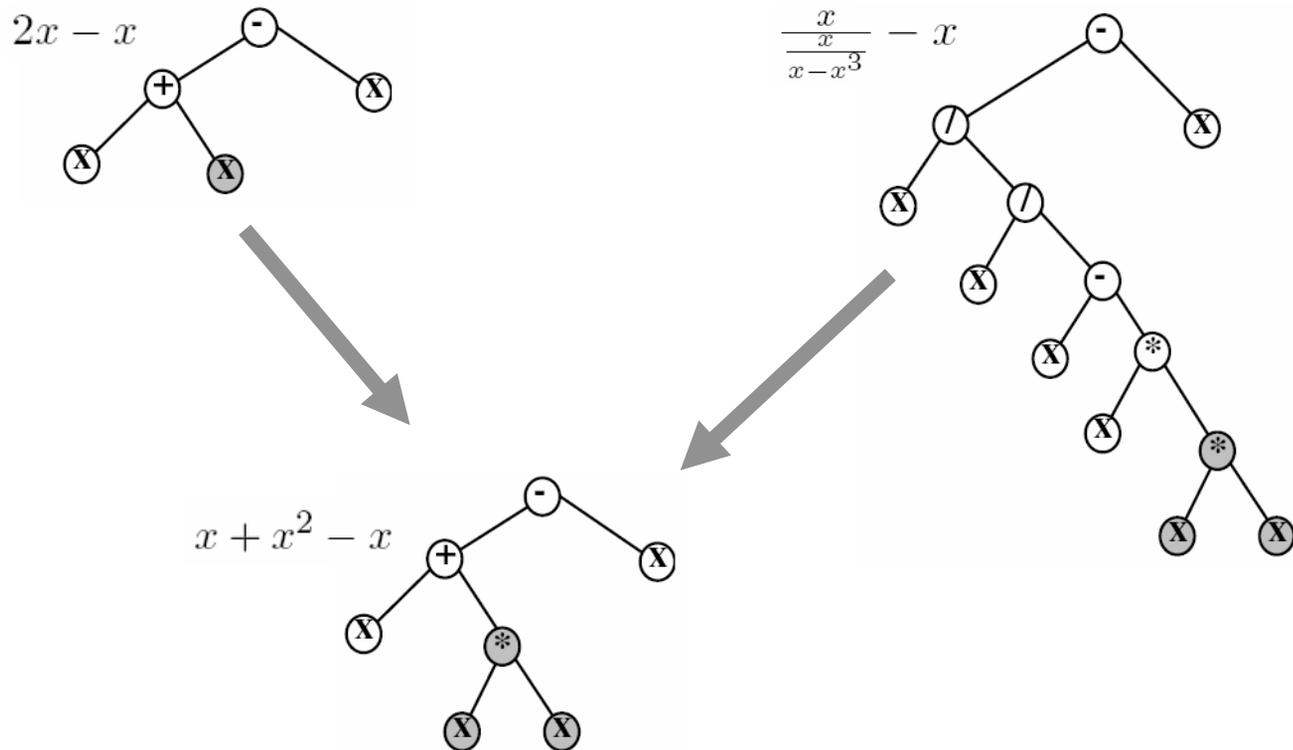
Robot: osservazioni

- I nodi dell'albero sono *typeless*
 - Questo facilita la generazione incrementale
- Non c'è passaggio di valori
 - La struttura dei nodi descrive solo il flusso dell'esecuzione
 - I sotto-alberi non passano un valore alla loro radice
- Non si usa memoria
 - Ogni ciclo sense-eval-act è un episodio indipendente
- Problemi per l'estensione
 - Il passaggio dei valori tende a introdurre la gestione dei *tipi*
 - Si consideri il caso: IF <sense> THEN <eval> ELSE <act>
 - La gestione della memoria può non essere semplice
 - Si può incorporare nei nodi ...

Esempio: symbolic regression

- Identificare l'espressione analitica di una funzione

– p.es $x + x^2 - x$



Symbolic regression: osservazioni

- I nodi dell'albero hanno un tipo
 - un valore reale (i.e. float o double)
 - identico per tutti i nodi (costanti, variabili operatori)
- La valutazione implica il passaggio di valori
 - I valori si propagano 'bottom-up'
 - I sotto-alberi passano un valore alla loro radice
- Non si usa memoria
- Cruciale è il metodo di valutazione della fitness