UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

Laurea triennale in Informatica
Anno accademico 2021/2022

Fondamenti di Intelligenza Artificiale

Lezione 9 - Algoritmi di ricerca locale (IV)

Algoritmi Genetici

Hands-on Algoritmi Genetici

E adesso? Vediamo come si progettano e implementano gli algoritmi genetici!

In realtà non ci serve null'altro di nuovo, bisogna soltanto implementare tutti i vari "building block" che formano un algoritmo genetico. Ricapitoliamoli:

Codifica Individui
Quale codifica si presta meglio per le soluzioni candidate?
Stringhe binarie/intere/reali o altre strutture dati?

Funzione/i Obiettivo
Problema mono/multi-obiettivo? Come si implementano le funzioni obiettivo? Le funzioni di valutazione coincidono con le funzioni obiettivo?

Operatori Genetici
Quale operatori genetici sono già definiti?
E' possibile implementare un operatore ad hoc per il mio problema?

- **Selezione**
  Quale algoritmo NON possiamo applicare? Usiamo l'**elitismo**?
  Se il problema è multi-obiettivo, quale **criterio di ordinamento** adottiamo?
- **Crossover**
  Quale algoritmo NON possiamo applicare?
  Quale probabilità di crossover? La probabilità si adatta a runtime?
- **Mutazione**
  Quale algoritmo NON possiamo applicare?
  Quale probabilità di mutazione? La probabilità si adatta a runtime?

Criteri di Arresto
Quali criteri usiamo? Quanto budget di ricerca possiamo allocare?
Disponiamo di un **test di ottimalità**?

Algoritmi Genetici

Hands-on Algoritmi Genetici

E adesso? Vediamo come si progettano e implementano gli algoritmi genetici!

In realtà non ci serve null'altro di nuovo, bisogna soltanto implementare tutti i vari "building block" che formano un algoritmo genetico. Ricapitoliamoli:

Popolazione — Quanto sono grandi le popolazioni?
La size si adatta a runtime?

Inizializzazione — Generiamo casualmente la prima popolazione?
Esistono delle euristiche per migliorare l'inizializzazione?

Vincoli — Il problema presenta dei vincoli?
Quale strategia usiamo per codificarli?

Meta-euristica — Come viene gestito l'algoritmo? L'evoluzione è generazionale o steady-state?
Se è multi-obiettivo, usiamo un **archivio**? Si applicano dei **miglioramenti locali**?

E' chiaro che per implementare da zero tutte queste cose serve tanto tempo. Quasi tutte queste componenti *molto generiche*, quindi una volta definite sono **riusabili per tutti i problemi** che vogliamo risolvere con algoritmi genetici (*problem-independent*).

Esiste qualcosa di **già pronto all'uso** che con *minime modifiche* ci permette l'implementazione di una GA?

Algoritmi Genetici

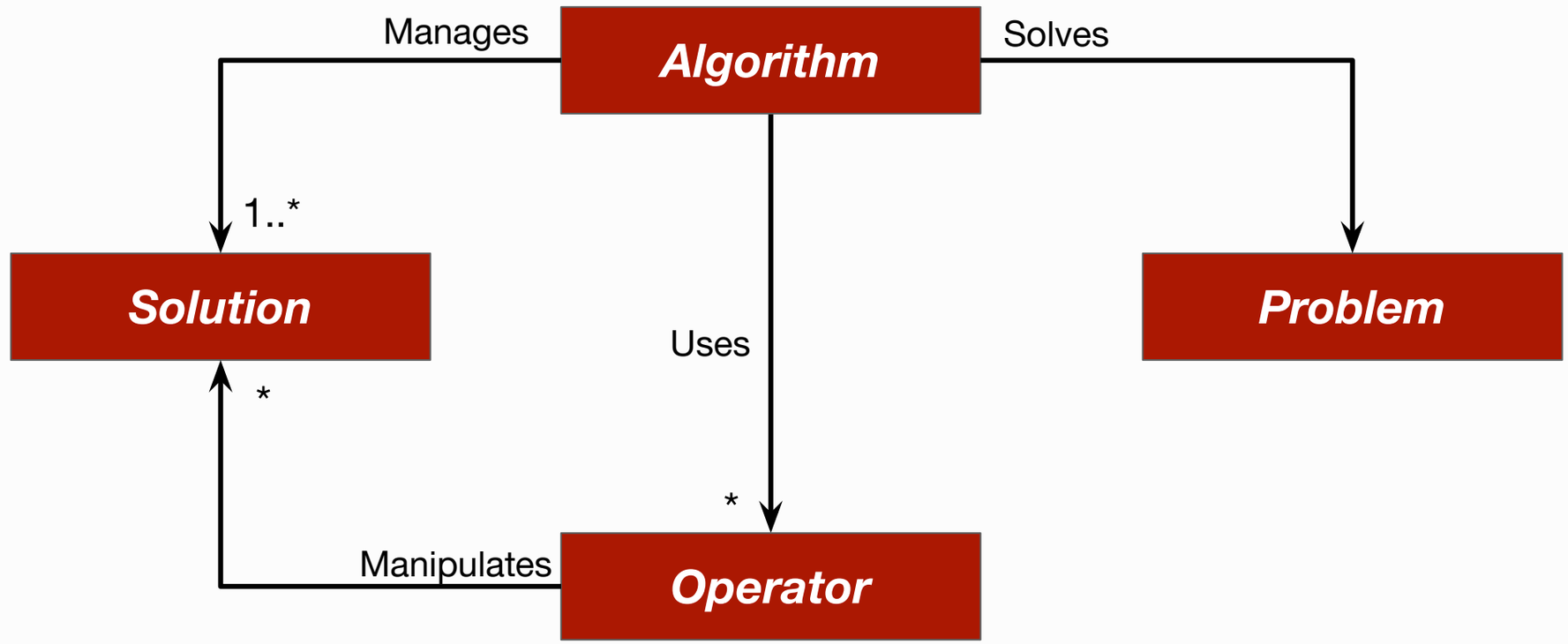
Hands-on Algoritmi Genetici

Framework open-source in Java per definire **meta-euristiche per risolvere problemi multi-obiettivo**. Alcune caratteristiche:

- Minimal-setup: è possibile definire un GA con poche righe di codice.
- Supporta i principali tipi di codifiche per le soluzioni.
- Non solo algoritmi genetici, ma anche *strategie evolutive* (ES) ed altre meta-euristiche.
- Le meta-euristiche note già implementate (SGA, MOGA, NSGA, ecc.).
- Alcuni algoritmi sono già resi paralleli.
- Implementa già alcuni problemi noti (ad esempio, TSP).
- E’ possibile ottenere statistiche di progresso durante l’esecuzione.

Algoritmi Genetici

Hands-on Algoritmi Genetici

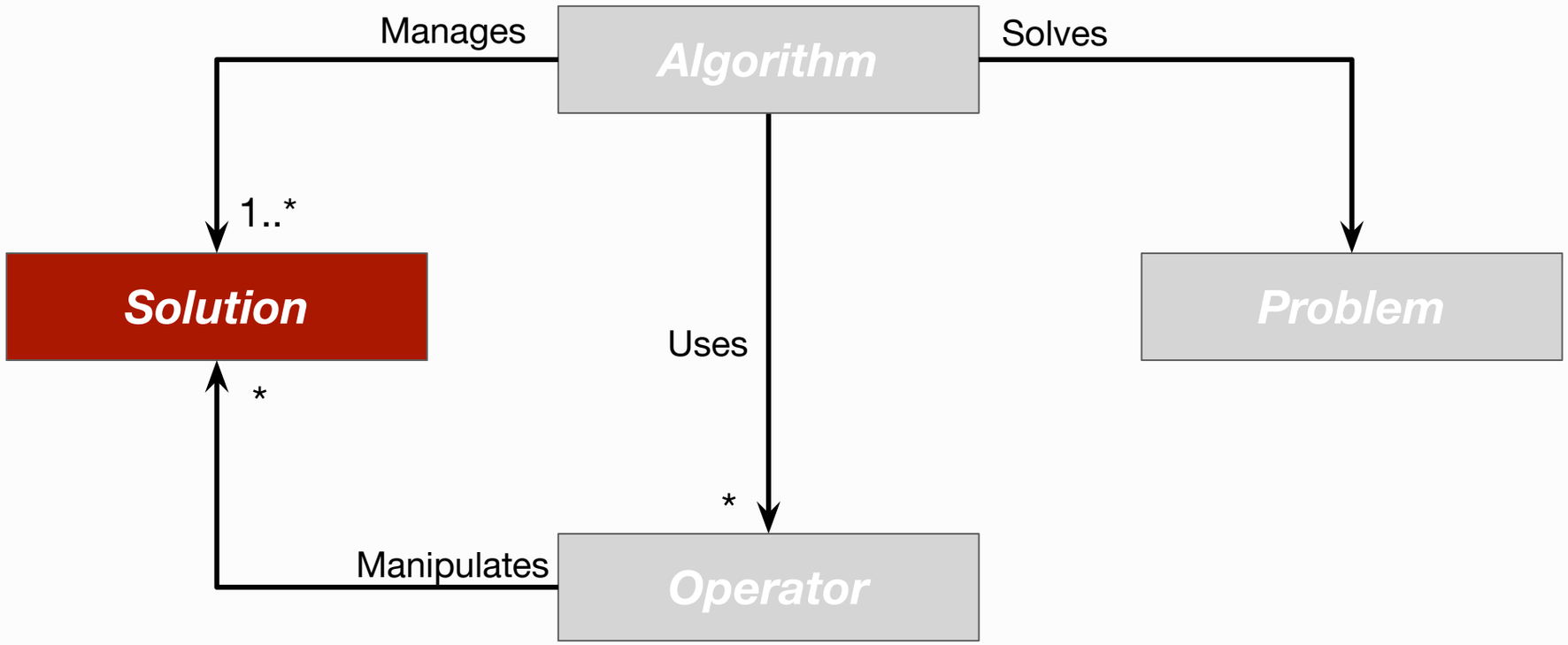


Questo class diagram rappresenta le principali **interfacce** di JMetal che bisogna conoscere necessariamente. Il suo funzionamento generale è così riassumibile:

> *"Un **Algoritmo** risolvere un **Problema** gestendo un insieme di **Soluzioni** candidate attraverso l'uso di diversi **Operatori**, che manipolano le Soluzioni"*

Algoritmi Genetici

Hands-on Algoritmi Genetici

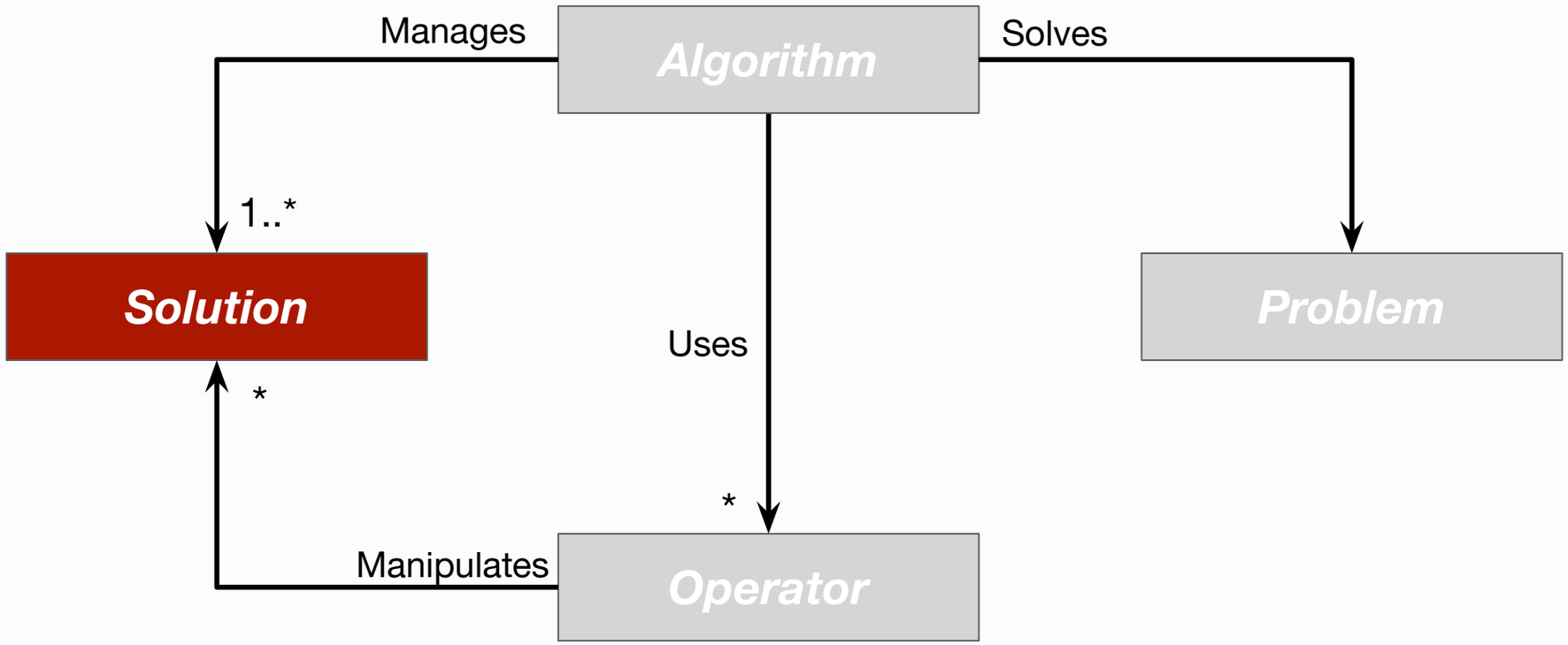


L'interfaccia *Solution* rappresenta una soluzione candidata (individuo negli algoritmi genetici). Supportando anche altre meta-euristiche, non si parla di *geni* ma di **variabili.** JMetal offre già alcune codifiche pronte all'uso:

- BinarySolution: stringhe binarie, con numero di variabili arbitrario
- IntegerSolution: stringhe intere, con numero di variabili arbitrario
- DoubleSolution: stringhe reali, con numero di variabili arbitrario
- SequenceSolution: stringhe di caratteri, con numero di variabili arbitrario
- PermutationSolution: stringhe rappresentanti una permutazione di interi
- ecc.

Algoritmi Genetici

Hands-on Algoritmi Genetici

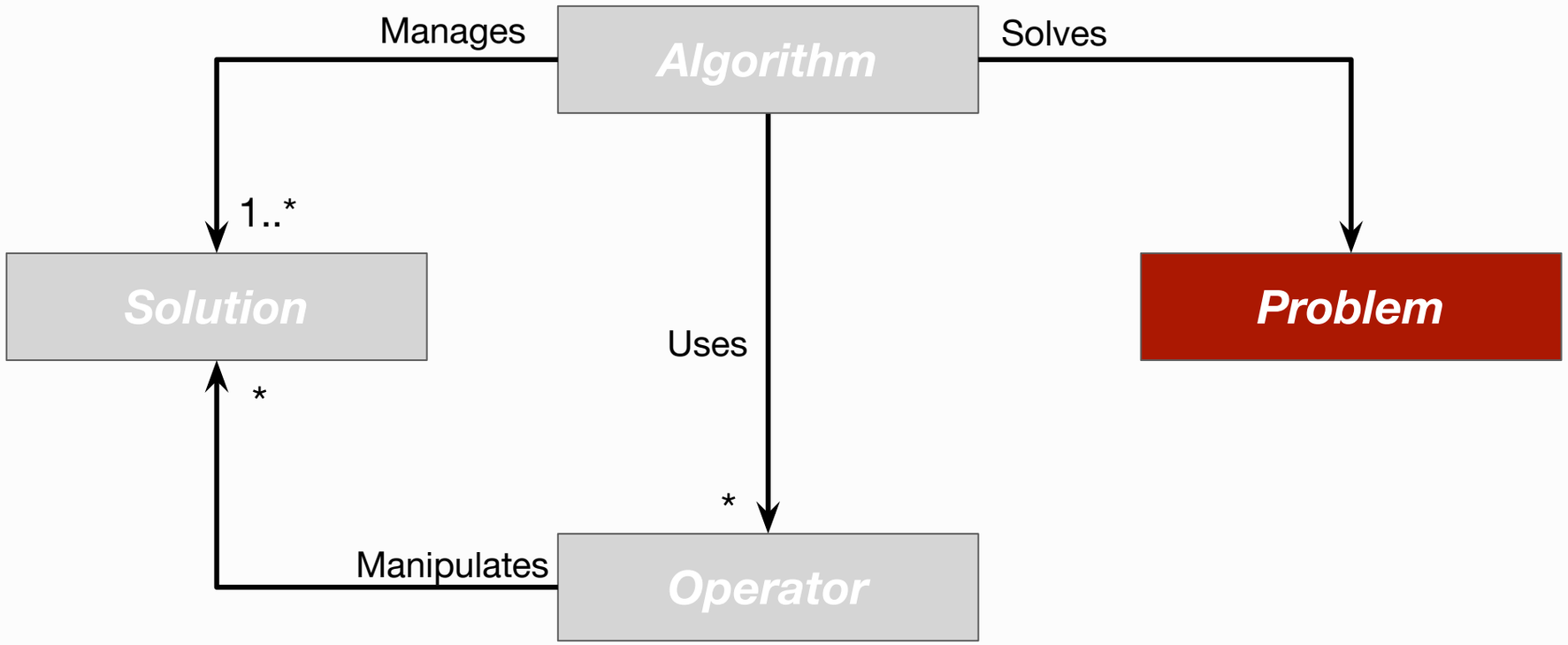


Una soluzione è formata da una **lista di variabili** (i geni per i GA), una **lista di valori obiettivo** (le valutazioni per i GA), una **lista di vincoli**, ed una **mappa di attributi** (per aggiungere informazioni extra). Inoltre, alcuni tipi di soluzioni, ad esempio quelle numeriche, possono essere **limitate** (bounded), stabilendo un range di valori che le variabili possono assumere (in questo modo si stabilisce il loro dominio).

E’ possibile creare copie con un metodo apposito.

Algoritmi Genetici

Hands-on Algoritmi Genetici



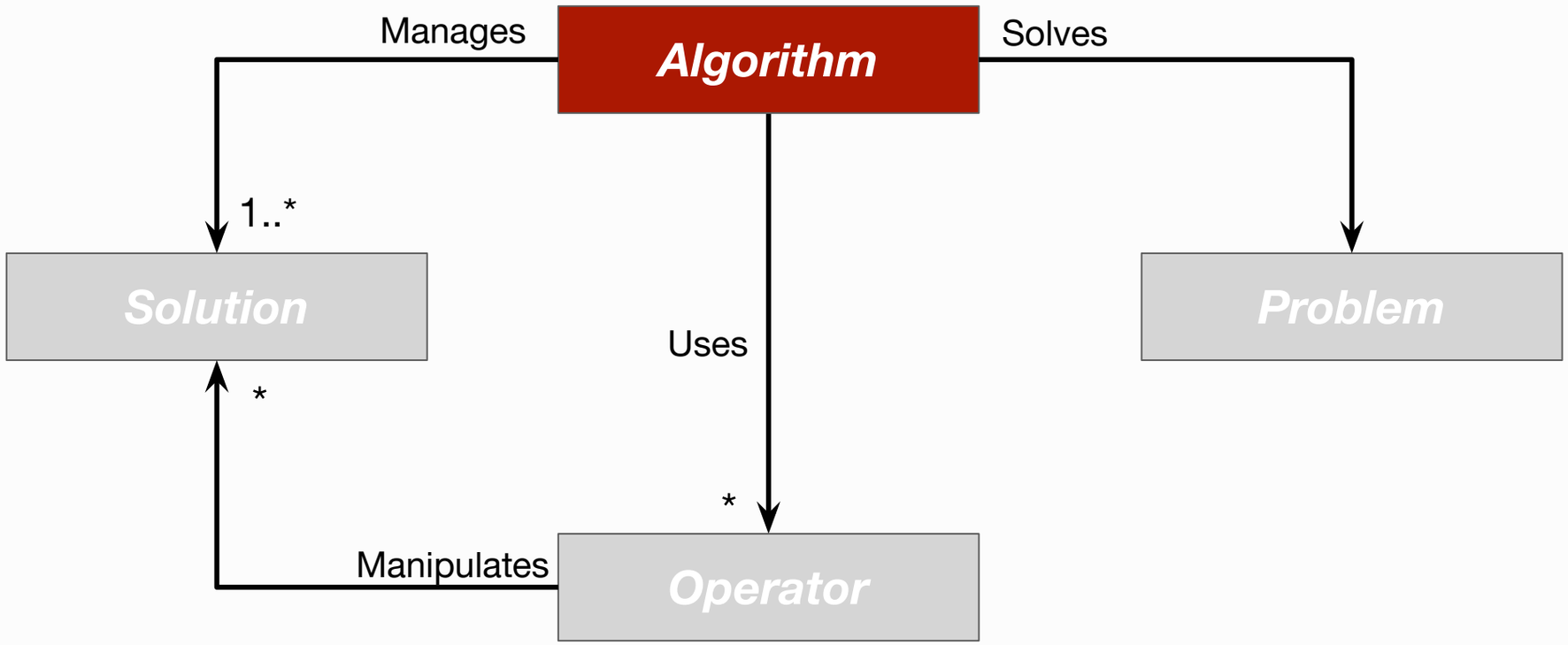
L'interfaccia *Problem* incarna un problema da risolvere (ad esempio, il problema del commesso viaggiatore o delle 8 Regine). E' l'interfaccia più importante che dobbiamo necessariamente realizzare per definire il nostro problema, nel quale stabiliamo:

- Il numero di variabili (geni) delle soluzioni;
- il numero di funzioni obiettivo;
- il numero di vincoli (eventuali);
- i bounds delle variabili (eventuali).
- come calcolare le funzioni di valutazione, implementando il metodo `evaluate()`.

Esistono dei sottotipi astratti già implementati, ad esempio *AbstractIntegerProblem*.

Algoritmi Genetici

Hands-on Algoritmi Genetici



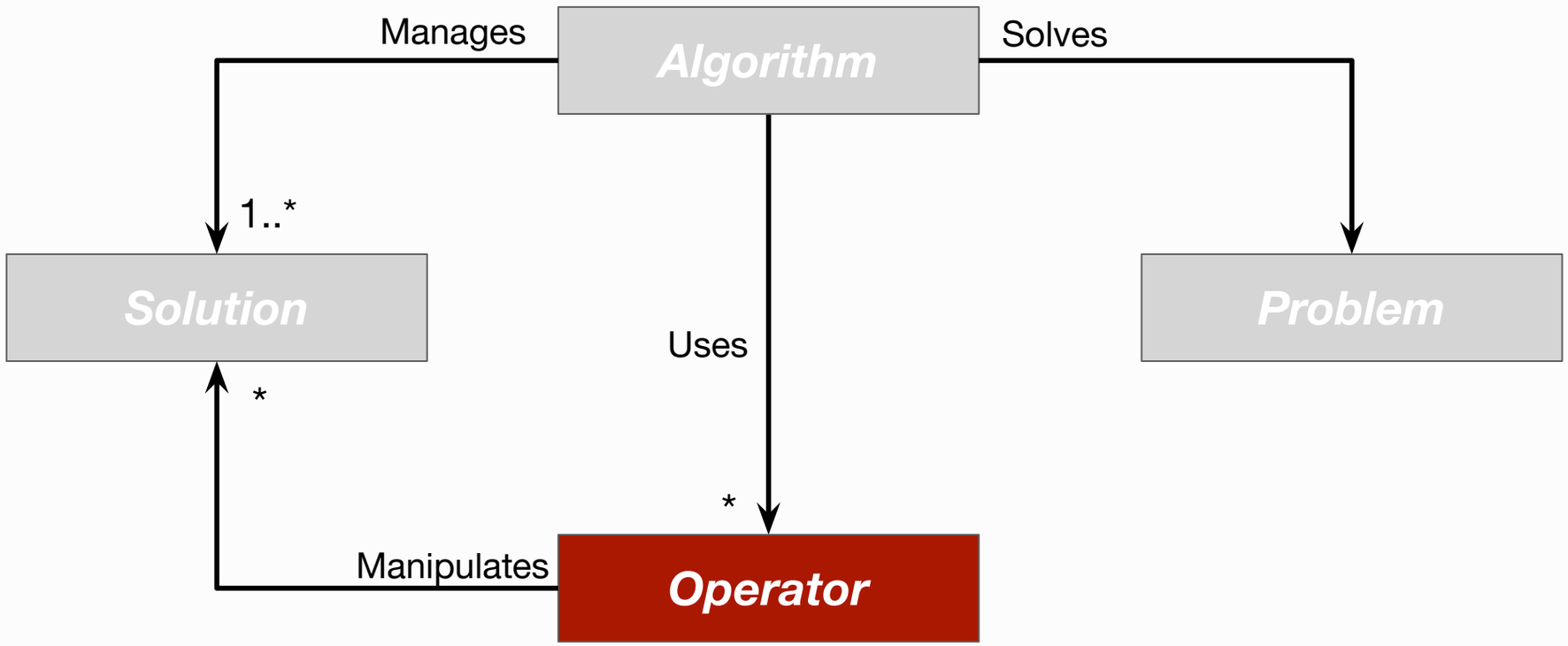
Un *Algorithm* corrisponde ad una specifica meta-euristica che vogliamo usare per risolvere un problema. Se vogliamo definire un nostro algoritmo genetico custom, allora dovremmo implementare una sottoclasse di *AbstractGeneticAlgorithm*. Se non ne abbiamo necessità, conviene utilizzare algoritmi già implementati. I più importanti sono:

- *GenerationalGeneticAlgorithm* (SGA)
- SteadyStateGeneticAlgorithm (Steady-State GA)
- *NSGAII*
- *SteadyStateNSGAII*

Per evitare di perdersi tra molte classi, JMetal offre delle classi *Builder* divise per tipo di algoritmo, ad esempio *GeneticAlgorithmBuilder* ed *NSGAIIBuilder*.

Algoritmi Genetici

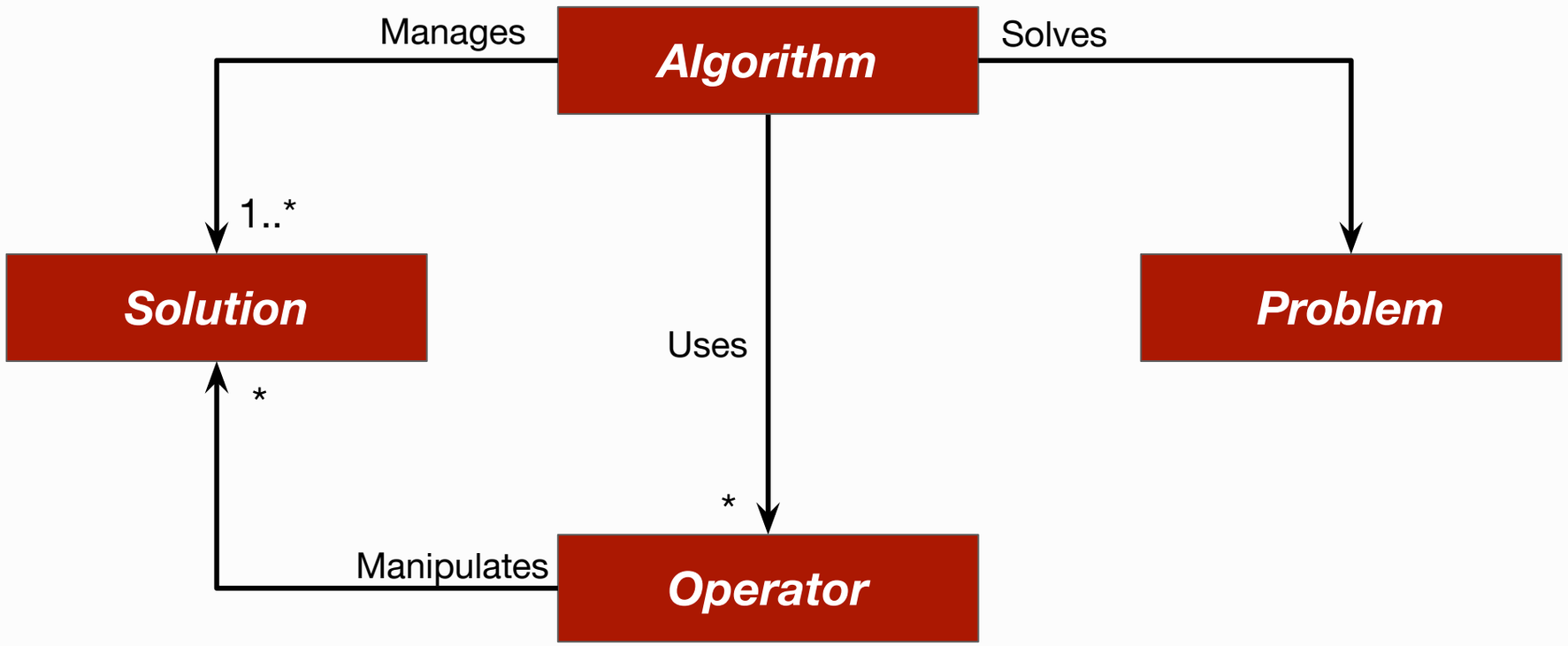
Hands-on Algoritmi Genetici



C’è poco da dire sull’interfaccia *Operator*. L’unica cosa è che **potrebbero non esserci tutti i tipi di operatori in JMetal**. Ad esempio, il *SinglePointCrossover* esiste ma funziona solo su *BinarySolution* (quindi usabili solo in sottoclassi di *BinaryProblem*), quindi per renderlo usabile per *IntegerProblem* dovremmo definire un *SinglePointCrossoverInteger* custom.

Algoritmi Genetici

Hands-on Algoritmi Genetici



In sintesi, questi sono i passi da seguire per definire un GA con minimal setup:

1. **definire il problema**, implementando una sottoclasse di *AbstractXProblem*, dove X può essere *Binary, Integer, Double*, ecc.
   a. Stabilire numero variabili, obiettivi, ed eventuali bound e vincoli;
   b. definire il calcolo della/e valutazione/i (implementare `evaluate()`);
2. **definire una classe con `main()`** che istanzia il problema definito, oltre che un operatore di selezione, uno di crossover e uno di mutazione;
3. **istanziare un algoritmo noto** (anche passando tramite uno dei *Builder*) consegnandogli tutti i parametri necessari, tra cui la taglia della popolazione, il numero massimo di valutazioni, ecc. (i parametri richiesti variano a seconda dell'algoritmo);
4. **lanciare l'esecuzione** passando per un'istanza di *AlgorithmRunner*;
5. **ottenere il risultato finale** dall'algoritmo.

Algoritmi Genetici

Problemi del Mondo Reale

Ingegneria del Software non è solo documenti… essa ha lo scopo di ingegnerizzare il software, renderlo un prodotto industriale soggetto a processi ben definiti.

In altri termini, **fare ordine** su tutti gli aspetti che riguardano lo sviluppo e la manutenzione del software, dal suo concepimento fino alla sua “morte”.

I problemi tipici dell’Ingegneria del Software riguardano **tutte le fasi del ciclo di vita del software** (Requisiti, Design, Implementazione, Testing, Deployment, Monitoraggio), ma possono riguardare anche **aspetti relativi al processo** (Stima dei Costi, Gestione delle Risorse, Prioritizzazione degli Interventi di Manutenzione).

Molti di essi “invocano” l’utilizzo di tecniche di ottimizzazione per essere risolti:

- Qual è il miglior set di requisiti che bilanciano il **costo** e la **soddisfazione** dei clienti?
- Qual è la **migliore allocazione** di task agli sviluppatori in base alle loro **capacità**?
- Qual è il **minimo set** di classi da ispezionare per **massimizzare i bug** identificati?
- Qual è il **miglior modo di organizzare** i sottosistemi dell’intero sistema?

Questi problemi si prestano molto bene agli algoritmi genetici per un semplice motivo: la **funzione obiettivo è quasi sempre accessibile e semplice da calcolare**, visto che spesso sono basate su metriche software facili da estrarre.

Questo ambito di ricerca è noto come **Search-based Software Engineering (SBSE)**.

Vediamo alcuni di questi problemi nel dettaglio e capiamo come formularli per un GA.

Algoritmi Genetici

Problemi del Mondo Reale - Next Release Problem (NRP)

Problem Statement

Immaginiamo di avere un sistema software già rilasciato e messo in esercizio. Durante il suo funzionamento, si ricevono molte *richieste di cambiamento*, magari provenienti da utenti finali (ad esempio, tramite feedback) o da clienti. Queste richieste, una volta analizzate, producono dei **nuovi requisiti**, magari così tanti che **non è possibile includerli tutti nella prossima release**, ma bisogna capire quali considerare e quali rimandare in una futura release.

Avrebbe senso capire non solo la **rilevanza** di ciascun requisito, ma anche il **costo** in termini di ore/persona. Sulla base di questi dati, possiamo individuare il **miglior insieme di requisiti da consegnare nella prossima release**.

Definiamo più formalmente questo problema.

Algoritmi Genetici

Problemi del Mondo Reale - Next Release Problem (NRP)

Formulazione

Input: Dato un insieme di n requisiti **indipendenti** $R = \{r_1, \ldots, r_n\}$ ed un insieme di m clienti $C = \{c_1, \ldots, c_m\}$ che hanno avanzato i requisiti in R.
Ad ogni requisito si associa un **costo** intero positivo tramite la funzione $cost(r)$, nota a priori. Ad ogni cliente si associa un **peso** (importanza per l'azienda) compreso tra 0 e 1 tramite la funzione $weight(c)$, anche essi noti a priori. La somma di tutti gli m pesi è 1. Ogni cliente mostra un **interesse** intero non negativo per ciascun requisito, catturato dalla funzione $value(r_i, c_j)$. Si definisce, quindi, una funzione $score(r_i)$ in grado di catturare l'**importanza del requisito** come la somma di tutti gli interessi pesata sull'importanza di ciascun cliente.

$$score(r_i) = \sum_{j=1}^{m} w_j \cdot value(r_i, c_j)$$

Output: Uno o più **subset di requisiti** tali che siano di:

- **Massima importanza (score) totale.**
- **Minimo costo totale**.

Hands-on: come modelliamo gli individui, la/le funzione/i obiettivo?

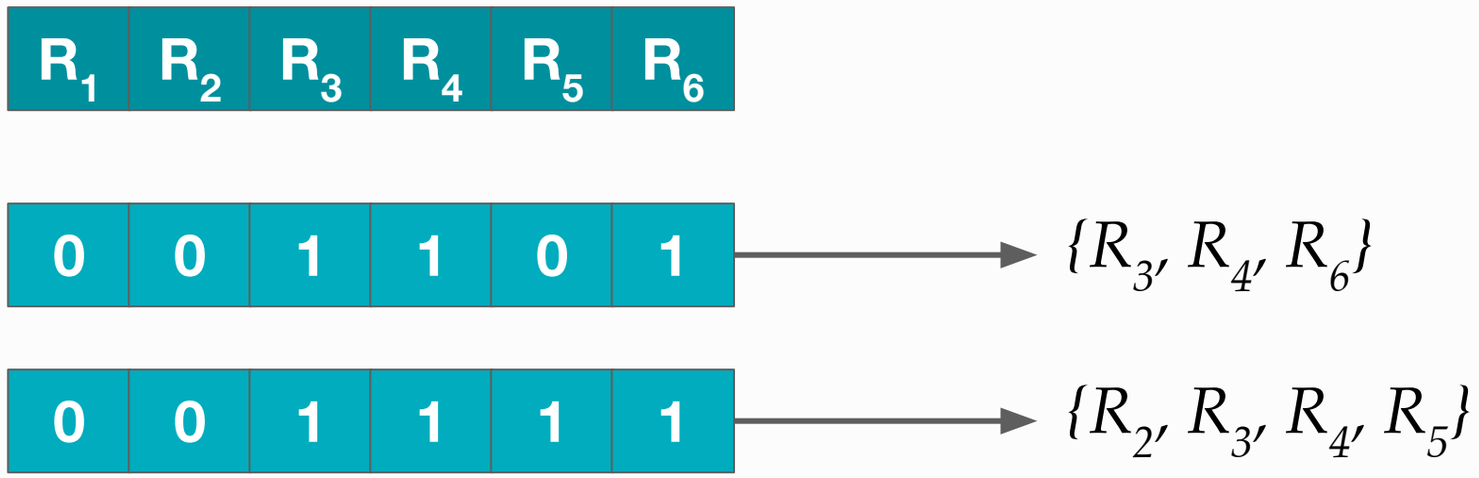
Algoritmi Genetici

Problemi del Mondo Reale - Next Release Problem (NRP)

Modellazione

Sappiamo che ad ognuno degli n requisiti è associato un numero intero, $i = 1, \ldots n$. Questa informazione possiamo usarla per selezionare in maniera compatta un sottoinsieme di requisiti utilizzando un **vettore logico**, ovvero un vettore booleano che compie una selezione degli elementi posti in corrispondenza a seconda del valore: 1 se si vuole selezionare il requisito i-esimo, altrimenti 0.

Ad esempio:



Ha senso la codifica binaria per queste soluzioni? Pensiamo a crossover e mutazione:

- I crossover classici sono validi perché producono nuovi subset di requisiti.
- Le mutazioni classiche sono valide perché producono nuovi subset di requisiti.

Algoritmi Genetici

Problemi del Mondo Reale - Next Release Problem (NRP)

Modellazione

Sia *score* che *cost* sono accessibili e semplici da calcolare, quindi possiamo usarle entrambe as-is per fare le valutazioni degli individui (e quindi per la loro selezione).

Possiamo adoperare due tipi di modellazione:

- *Mono-obiettivo*: aggreghiamo le due funzioni (*score* e *cost*) con pesi 0.5 (nessuna preferenza). Bisogna, però, **negare la funzione *cost*** affinché la funzione aggregata sia massimizzabile. In questo caso, l'algoritmo restituisce *un singolo subset di requisiti sub-ottimale*.

$$\sum_{i=1}^{n} \frac{score(r_i) - cost(r_i)}{2} \cdot x_i$$

- *Multi-obiettivo*: massimizzare lo *score* totale, minimizzare il *cost* totale. In questo caso, l'algoritmo restituisce *un insieme di subset di requisiti sub-ottimali*.

$$maximize \sum_{i=1}^{n} score(r_i) \cdot x_i$$

$$minimize \sum_{i=1}^{n} cost(r_i) \cdot x_i$$

Commenti finali: Un problema che segue questo schema (ovvero, "identificare il **miglior subset di un insieme di elementi**") non è raro da trovare, e potremmo chiamarlo **"problema di selezione"**. In questi casi, gli individui si codificano come **stringhe binarie** per realizzare la selezione degli elementi.

Algoritmi Genetici

Problemi del Mondo Reale - Task Allocation

Problem Statement

Immaginiamo di far parte di un grande team di sviluppo assegnato ad un grande progetto software. Ad un certo punto arriva un grande carico di lavoro. Uno dei manager riesce ad analizzare tutto il lavoro, e individua un **grande insieme di task** fortunatamente **tutti indipendenti** (una semplificazione della realtà). Il manager è riuscito anche ad usare un ottimo *modello di stima dei costi*, ottenendo delle stime molto affidabili sull'**effort** richiesto da ciascun task, misurato in ore/persona. Ad ogni task può lavorare più di uno sviluppatore (questo significa che il numero di ore richieste per completare un task diminuisce se ci lavorano più persone insieme).

Si vuole stabilire un'allocazione ottimale di tutti i task tra gli sviluppatori del team, con lo scopo di **minimizzare il costo totale** (ovvero, terminare in meno ore possibile), e **minimizzare il carico di lavoro medio** (ore di lavoro) e **minimizzare il grado di multitasking medio** (numero di task assegnati)di ciascuno sviluppatore.

Hands-on: Formulare il problema e modellare la base di un algoritmo genetico (codifica individui e funzioni obiettivo/valutazione) in grado di ottimizzarlo.

Algoritmi Genetici

Problemi del Mondo Reale - Task Allocation

Formulazione

- **Task**: $T = \{t_1, \ldots, t_n\}$, dove $t_i.effort$ è un intero positivo che indica le ore/persone richieste.
- **Sviluppatori**: $D = \{d_1, \ldots, d_m\}$
- **Sviluppatori Allocati**: $alloc(t_i)$ restituisce un insieme di interi tra 1 ed m che corrisponde agli sviluppatori allocati su t_i.
- **Costo Task**: $costo(t_i) = t_i.effort / |alloc(t_i)|$
- **Task Assegnati**: $works(d_j)$ restituisce un insieme di interi tra 1 ed n che corrisponde ai task assegnati allo sviluppatore d_j.
- **Carico di Lavoro**: $\sum_{i \in works(d_j))} costo(t_i)$
- **Multitasking**: $multitasking(d_j) = |works(d_j)|$

Modellazione

- **Codifica**: Un individuo è un **vettore di insiemi di interi lungo n (numero di task)**. Il gene i-esimo contiene l'insieme degli sviluppatori assegnati al task t_i (ovvero $alloc(t_i)$).
- **Obiettivi**:

$$minimize \sum_{i}^{n} costo(t_i)$$

$$minimize \sum_{j}^{m} \frac{workload(d_j)}{m}$$

$$minimize \sum_{j}^{m} \frac{multitasking(d_j)}{m}$$

Commenti finali: Un problema che segue questo schema (ovvero, "identificare la **miglior allocazione** di risorse rispetto a delle entità") non è raro da trovare, e potremmo chiamarlo **"problema di allocazione"**. In questi casi, gli individui si codificano come **stringhe di insiemi di interi** per realizzare l'allocazione.

UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

Laurea triennale in Informatica
Anno accademico 2021/2022

Fondamenti di Intelligenza Artificiale

Lezione 9 - Algoritmi di ricerca locale (IV)