Una panoramica su algoritmi di ottimizzazione

Michele Ferraro, matricola 1717025

19 Agosto 2018

Abstract

Durante il corso di Intelligenza Artificiale è stata offerta una panoramica dei problemi di ottimizzazione, descrivendo cosa sono e alcuni algoritmi utili per trattarne la risoluzione. Con questo progetto si vuole ampliare la trattazione, offrendo una visione più ampia e affrontando nuove tipologie di algoritmi, nella fattispecie algoritmi di swarm intelligence e multiobiettivo. Oltre alla presente relazione, è disponibile un piccolo framework per l'implementazione di algoritmi metaeuristici, con alcuni algoritmi già pronti.

Introduzione

Formalmente [1], un problema di ottimizzazione ha la forma

$$\begin{aligned} &\underset{x}{\text{minimize}}\, f(x) \\ &g_i(x) \leq 0, \quad i = 1, \ldots, m \\ &h_j(x) = 0, \quad j = 1, \ldots, p \end{aligned}$$

dove $f(x)$ è la funzione (a codominio reale) il cui valore vogliamo minimizzare (in questo caso viene generalmente detta "funzione di costo"), x è il vettore dei suoi parametri, m e p sono interi positivi, e g e h sono famiglie di funzioni che identificano i vincoli che i parametri devono rispettare. È anche possibile una forma alternativa in cui si vuole massimizzare il valore della funzione obiettivo.

Proponiamo di seguito alcuni semplici esempi di problema di ottimizzazione per vedere nella pratica il senso della definizione.

- **Esempio 1** (un problema banale) - Trovare il valore di x, $x \in \mathbb{R}$, che minimizza la funzione $f(x) := x^2$ (è una parabola, con ovvia soluzione $x = 0$).

- **Esempio 2** (un problema leggermente più complesso e vincolato) - Trovare il rettangolo di area massima, dato il suo perimetro p (il problema si risolve facilmente per via analitica, ed è usato solamente come esempio immediato). Le variabili di ottimizzazione sono r_w e r_h, rispettivamente larghezza e altezza del rettangolo. La funzione obiettivo, in questo caso da massimizzare, è $f(r_w, r_h) := r_w \cdot r_h$ (ci si può facilmente riportare alla definizione originaria, con la funzione obiettivo da minimizzare, utilizzando come funzione obiettivo $f'(r_w, r_h) \coloneqq -f(r_w, r_h)$). Questo problema presenta un solo vincolo da soddisfare, $2 \cdot (r_w + r_h) = p$. Nella notazione originale, dobbiamo avere $h_1(r_w, r_h) = 0$, con $h_1(r_w, r_h) \coloneqq 2 \cdot (r_w + r_h) - p$. Soluzione ottima del problema è $r_w = r_h = \frac{p}{4}$ (fissato il perimetro, il rettangolo di area massima è un quadrato).

- **Esempio 3** (un problema reale) - Si dispongono di risorse r_1, ..., r_{n_r} in quantità a_1, ..., a_{n_r}. Le risorse possono essere utilizzate nella realizzazione di prodotti p_1, ..., p_{n_p}, ognuno dei quali necessità delle quantità (q_1^i, ... $q_{n_r}^i$ e può essere venduto per un guadagno di g_1, ..., g_{n_r}. Si vogliono quindi allocare le risorse disponibili alla produzione di prodotti (uno stesso prodotto può essere realizzato anche più volte se le risorse disponbili lo consentono) in modo da massimizzare il guadagno della vendita di tutti i prodotti.

A seconda delle caratteristiche dei parametri o delle funzioni coinvolte, è possibile suddividere i problemi di ottimizzazione in diverse categorie: In base al dominio dei parametri, ad esempio, si distingue tra problemi interi (tutti i parametri a dominio intero), continui (tutti i parametri a dominio reale) e misto-interi (alcuni parametri a dominio intero e altri a dominio reale).
In base alla presenza o meno di vincoli sui parametri si parla di problemi vincolati e non vincolati. Un problema può avere una sola funzione obiettivo oppure più di una, in questo caso si dice multiobiettivo.

Un apporto sostanziale alla risoluzione di problemi di ottimizzazione viene dall'analisi matematica, ma le tecniche di questa categoria non saranno trattate nel dettaglio. Molto brevemente, la conoscenza della forma delle funzioni obiettivo o dei vincoli permette di trovare soluzioni mediante metodi analitici.

Algoritmi di ottimizzazione

È possibile dividere gli algoritmi di ottimizzazione in due macro-categorie, in base alla metodologia di risoluzione. In primo luogo troviamo algoritmi basati sul calcolo del gradiente. Appartiene a questa categoria, ad esempio, l'algoritmo di steepest ascent/descent. Per problemi che ne rispettano le condizioni, metodi molto efficienti sono stati sviluppati per trovare soluzioni. Sono tuttavia meno consigliabili in caso di funzioni più complesse, non differenziabili, con discontinuità, o con presenza di ottimi locali.

Ci concentriamo invece sugli algoritmi di risoluzione meta-euristici. Per meta-euristica si intende, secondo la definizione in [2], una procedura di livello superiore, indipendente dal problema specifico, che fornisce linee guida o strategie da utilizzare nello sviluppo di algoritmi di ottimizzazione euristici. Più semplicemente è, come suggerisce il nome, un'euristica per la gestione o selezione di euristiche. Una caratteristica di spicco di tali algoritmi è che, non necessitando del gradiente e, più in generale, facendo assunzioni minime sulla forma del problema, possono essere facilmente applicabili a una vasta gamma di problemi. Appartengono a questa categoria, tra gli altri, gli algoritmi di swarm intelligence ed evolutivi. Tra i contro, questi algoritmi si affidano spesso alla casualità e non danno garanzie di ottimalità della soluzione trovata.

Ottimizzazione multiobiettivo e SPEA2

Gli algoritmi visti finora lavorano per minimizzare o massimizzare il valore di una sola funzione obiettivo. Altri algoritmi si occupano di massimizzare più obiettivi contemporaneamente. Problemi con più funzioni obiettivo sono detti multiobiettivo. Un principio importante per questa categoria di proble-

mi è che spesso non esiste una soluzione ottima, cioè migliore di ogni altra rispetto a tutti gli obiettivi. Per questo motivo, c'è bisogno di compromessi nella scelta di una soluzione, e algoritmi di questa categoria calcolano non una sola soluzione, ma un insieme di soluzioni, tali che, comunque siano prese due soluzioni dall'insieme, nessuna delle due sia migliore dell'altra rispetto a tutti gli obiettivi. Formalmente, prese due soluzioni s_1 e s_2 e una famiglia $f_i, i = 1, \ldots, n$ di funzioni obiettivo da massimizzare, si dice che s_1 domina s_2, e si scrive $s_1 \succ s_2$, se

$$\underset{i \in 1,\ldots,n}{\forall} f_i(s_1) \geq f_i(s_2) \quad \wedge \quad \exists i \in 1, \ldots, n : f_i(s_1) > f_i(s_2).$$

Si definisce ottimo paretiano una soluzione non dominata da nessun'altra. Algoritmi come SPEA2 [3], che si descrive di seguito, si prefiggono lo scopo di trovare il fronte di Pareto, composto dall'insieme delle soluzioni pareto-ottimali.

Lo Strength Pareto Evolutionary Algorithm 2 è un algoritmo che presenta tutte le caratteristiche principali degli algoritmi evolutivi (riproduzione, crossing over, mutazioni), con particolarità che lo rendono adatto alla ricerca multiobiettivo. Un punto interessante dell'algoritmo è che, oltre alla popolazione di ogni generazione, è presente un secondo insieme di individui, detto archivio, che mantiene individui non dominati attraverso le successive iterazioni/generazioni (si parla in questo caso di elitismo). Inoltre, la scelta degli individui da usare per le operazioni di riproduzione avviene tra i soli membri dell'archivio, limitando il ruolo della popolazione di una generazione a pool di candidati ad una posizione dell'archivio nella generazione successiva.

Input:

N_P : dimensione della popolazione
N_A : dimensione dell'archivio
T : numero di generazioni

Output:

S : l'insieme delle soluzioni pareto-ottimali trovate

Algoritmo:

1) Crea P_0, popolazione di N_P individui casuali, e $A_0 = \emptyset$, archivio inizialmente vuoto. Imposta $t = 0$.

2) Calcola i valori delle funzioni obiettivo per tutti gli individui in P_t e A_t, e conseguentemente i valori di fitness come segue:
ad ogni individuo i di P_t e A_t associa un valore S (*strength*):

$$S(i) = |\{j \in P_t \cup A_t : i \succ j\}|.$$

La *strength* di un individuo corrisponde al numero di individui che domina nell'insieme di riferimento ($P_t \cup A_t$). Sulla base di questo, associa ad ogni individuo un valore di *raw fitness* R:

$$R(i) = \sum_{j \in P_t \cup A_t : j \succ i} S(i),$$

corrispondente alla somma dei valori di *strength* che dominano l'individuo. Per ogni individuo i, infine, si assegna una penalità basata sulla densità di distribuzione delle soluzioni: sia σ_i^k la distanza tra il vettore dei parametri di i e quello del k-esimo individuo ad esso più vicino in $P_t \cup A_t$ (generalmente si sceglie $k = \sqrt{N_P + N_A}$). Si definisce quindi la densità di di un individuo come:

$$D(i) = \frac{1}{\sigma_i^k + 2}.$$

Tale penalità viene aggiunta per incoraggiare una distribuzione uniforme delle soluzioni lungo il fronte di Pareto.
A questo punto la fitness (da minimizzare) F di un individuo i è semplicemente $F(i) = R(i) + D(i)$.
Per la definizione di D, considerando che σ_i^k è un reale non negativo, $D(i) \in (0, 1)$ e funge quindi da spareggio tra soluzioni con valori di *raw fitness* equivalenti.

3) Crea $A_{t+1} = \emptyset$, l'archivio della generazione successiva. Copia in A_{t+1} tutti gli individui di P_t e A_t che non sono dominati. In altre parole, $A_{t+1} \leftarrow \{i \in P_t \cup A_t : (\nexists i' \in P_t \cup A_t : i' \succ i)\}$. È possibile eseguire questa operazione copiando gli individui i tali che $F(i) < 1$. Se $|A_{t+1}| > N_A$, elimina gli elementi in eccesso tramite l'operazione di troncamento (passo 3.1), altrimenti aggiungi a A_{t+1} elementi dominati (passo 3.2).

3.1) Per riempire l'insieme A_{t+1}, è sufficiente ordinare l'insieme $P_t \cup A_t$ in ordine crescente di fitness. A questo punto basta copiare i primi $N_A - |A_{t+1}|$ elementi con fitness ≥ 1 (quelli con fitness inferiore a 1 sono già stati inseriti in quanto non dominati da nessuno).

3.2) Si definisce una relazione d'ordine debole su A_{t+1}: dati due suoi individui i e j, diciamo che:

$$i \leq_d j \iff$$
$$\forall 0 < k < |A_{t+1}| : \sigma_i^k = \sigma_j^k \quad \vee$$
$$\exists 0 < k < |A_{t+1}| : \left[\left(\forall 0 < l < k : \sigma_i^l = \sigma_j^l \right) \wedge \sigma_i^k < \sigma_j^k \right]$$

In parole, i precede j nell'ordinamento se i valori di σ_i e σ_j sono uguali per ogni k (e in questo caso vale anche $j \leq_d i$), oppure se, per il primo k per cui tali valori differiscono, si ha $\sigma_i^k < \sigma_j^k$ (in questo caso $i <_d j$). In questo caso, i valori σ vengono calcolati similarmente agli analoghi nel punto 2 dell'algoritmo, ma sul solo insieme A_{t+1}. A questo punto si cerca un minimo di questo ordinamento, cioè un elemento
$i \in A_{t+1} : \forall j \in A_{t+1} \quad i \leq_d j$, e lo si rimuove da A_{t+1}. Si cerca in questo modo di eliminare agglomerati molto concentrati di soluzioni, lasciando invece intatti individui più isolati. Questo punto viene reiterato finché non si arriva ad ottenere $|A_{t+1}| = N_A$.

4) Se si raggiunge il criterio di terminazione (t = T), restituisci l'insieme S di soluzioni in A_{t+1} non dominate.

5) Riempi il mating pool con elementi del solo archivio A_{t+1}, scelti tramite tornei binari con reinserimento.

6) Applica le operazioni di crossover e mutazione agli elementi della mating pool fino a ottenere N_P individui, che costituiscono P_{t+1}, la popolazione della generazione successiva. Aumenta t di 1 vai al passo 2.

Swarm intelligence e Particle Swarm Optimization

Negli algoritmi di ricerca locale, la procedura prevede la modifica graduale di uno stato (un assegnamento alle variabili) per arrivare, passo dopo passo, alla soluzione finale. Negli algoritmi di swarm intelligence, invece, si mantengono contemporaneamente più soluzioni, che possono "interagire" per dar vita a comportamento intelligente. Appartengono a questa categoria numerosi algoritmi di ispirazione naturale, che riescono ad ottenere buoni risultati imitando il comportamento di gruppi di animali (ad esempio artificial bee colony e ant colony optimization). Tali algoritmi sono però stati criticati perché, per quanto utili, hanno raramente portato innovazioni nel campo dell'intelligenza artificiale o dell'ottimizzazione, limitandosi ad essere esperi-

menti pratici ma senza elementi di novità a livello teorico. [4]

Si descrive di seguito un algoritmo di questo tipo, Particle Swarm Optimization [5]. Ragioniamo in termini di funzione obiettivo da massimizzare. L'algoritmo prevede diverse istanze di particelle, ognuna delle quali memorizza posizione e velocità correnti e la migliore posizione individualmente attraversata. Si memorizza anche la migliore posizione tra tutte quelle iniziali delle particelle. Ad ogni iterazione, la velocità di ogni particella viene modificata, tendendo a muovere la particella verso gli ottimi individuale e globale.

Input:
x_{min}, x_{max} : bound per i parametri
N : numero di particelle
ω : fattore di inerzia
ϕ_i : fattore cognitivo
ϕ_s : fattore sociale
I : numero di iterazioni

Output:
S : la migliore soluzione trovata

Algoritmo:

1) Inizializza le N particelle in posizioni (assegnamenti di parametri) casuali. Per ognuna, imposta il valore migliore attraversato alla posizione corrente. Imposta il valore del massimo globale *best_pos* alla posizione della particella a fitness maggiore.

2) Per ogni particella, imposta la velocità corrente a un vettore di valori casuali uniformi compresi tra $-|x_{max} - x_{min}|$ e $|x_{max} - x_{min}|$. La velocità di una particella lungo una dimensione è quindi al più, in valore assoluto, l'intera ampiezza dell'intervallo ammissibile per la coordinata corrispondente.

3) Per ogni particella p, aggiorna il valore della velocità $p_{velocity}$ secondo la formula
$p_{velocity} \leftarrow \omega \cdot p_{velocity} + \phi_i \cdot r_i \cdot (p_{best\_pos} - p_{curr\_pos}) + \phi_s \cdot r_s \cdot (best\_pos - p_{curr\_pos})$,
dove r_i e r_s sono vettori di reali con distribuzione uniforme in $(0, 1)$, campionati per ogni particella. Tale formula è assimilabile a una combinazione lineare della precedente velocità e delle distanze dalle posizioni migliori individuale e globale, i cui coefficienti sono parametri dell'algoritmo, e con aggiunta di un elemento di casualità nei vettori r_i e r_g. La particella quindi

si muove inizialmente in una direzione casuale, ma la sua traiettoria viene influenzata nel tempo da p_{best_pos} e $best\_pos$.

4) Aggiorna la posizione di ogni particella p, secondo la formula $p_{curr\_pos} \leftarrow p_{curr\_pos} + p_{velocity}$, cioè semplicemente sommando posizione e velocità correnti. Se la nuova posizione è migliore della posizione migliore attraversata dalla particella (cioè se $fitness(p_{curr\_pos}) > fitness(p_{best\_pos})$), aggiorna il valore di p_{best_pos} ($p_{best\_pos} \leftarrow p_{curr\_pos}$). Se la nuova posizione è anche migliore della migliore posizione globale aggiorna anche quest'ultima (quindi, $best\_pos \leftarrow p_{curr\_pos}$).

5) Se sono state eseguite I iterazioni (o comunque si verifica una condizione di terminazione), restituisci $best\_pos$, altrimenti ritorna al punto 3.

PSO, varianti e lo stato dell'arte

Il funzionamento della versione base dell'algoritmo PSO è stato esteso in numerosissime versioni, in cui guadagna la possibilità di essere applicato a problemi di ottimizzazione (misto-)intera, vincolati, multiobiettivo. Ampio spazio è stato dato alla ricerca di valori e procedure di metaottimizzazione per i parametri dell'algoritmo. Una buona analisi dello stato dell'arte è disponibile in [6].

Una prima alternativa (Fully Informed PSO) vede la velocità della particella essere influenzata dalle posizioni migliori di tutte le altre particelle, non viene cioè mantenuto l'ottimo globale. Formalmente, nella formulazione originale [7], in fase di aggiornamento della velocità, date costanti ϕ (costante di accelerazione, con ruolo analogo a ϕ_i e ϕ_s nella descrizione dell'algoritmo data precedentemente) e χ, la velocità della particella p viene aggiornata secondo la formula

$$p_{velocity} \leftarrow \chi \left(p_{velocity} + \sum_{p' \in Particles} \frac{\phi * Uniform(0,1) * \left(p'_{best\_pos} - p_{curr\_pos}\right)}{N} \right).$$

Varianti intermedie tra i due approcci prevedono invece clustering delle particelle, ognuna delle quali è influenzata soltanto dalle particelle (o dalla migliore delle particelle) nel suo vicinato. In [8], le particelle e la relazione di vicinanza nell'insieme delle stesse viene espresso tramite un grafo, i cui nodi corrispondono alle particelle e c'è un arco due nodi-particelle se le due si in-

fluenzano a vicenda. In questa formulazione, il grafo della variante canonica dell'algoritmo è completamente connesso; varianti includono disposizioni a griglia (ogni particella connessa agli adiacenti nella griglia) e ad anello (ogni particella connessa alle due particelle che la precedono e seguono).

Un elemento di ricerca è l'equilibrio tra strategie focalizzate su exploration (esplorare porzioni dello spazio di ricerca non ancora visitate per scoprire potenziali nuove buone posizioni) e exploitation (esplorare le vicinanze di una buona posizione già trovata). In [9], si propone una strategia adattiva per i parametri ω, ϕ_i e ϕ_s in funzione della densità delle particelle: ad ogni iterazione, si calcola la distanza media di ogni particella p secondo la seguente formula:

$d_p \coloneqq \frac{1}{N-1} \sum_{p' \in Particles, p' \neq p} \left\| p_{curr\_pos} - p'_{curr_pos} \right\|$, cioè semplicemente come media delle distanze rispetto a tutte le altre particelle.
Si calcolano d_g come il d della particella correntemente nella posizione migliore, d_{min} e d_{max} come i d minimi e massimi, e da questi si ricava il fattore evolutivo

$$f \coloneqq \frac{d_g - d_{min}}{d_{max} - d_{min}} \in [0, 1].$$

f indica quindi dove la densità della particella migliore si colloca rispetto all'intero intervallo delle densità di tutte le particelle. Un valore intermedio di f indica uno stato di esplorazione (non ci sono agglomerati significativi di particelle), un valore basso una fase di convergenza (la particella migliore si trova in una porzione ad alta densità di particelle), mentre un valore alto indica la probabile presenza di una concentrazione di particelle in un ottimo locale da cui bisogna sfuggire (questa fase viene indicata con il nome di *Jumping Out*). In base al valore di f si incrementano o decrementano l'inerzia e i fattori cognitivo e sociale da usare nello sciame. In fase di Jumping Out, ad esempio, diminuisce ϕ_i e aumenta ϕ_s, in modo che le particelle intrappolate nell'ottimo locale tendano a fuggirne, in direzione dell'ottimo esterno. Analoghi comportamenti si verificano per le altre fasi.
Un'ampia area di ricerca è rappresentata dagli algoritmi ibridi, che cioè uniscono alla PSO procedimenti tipici di altri algoritmi di ottimizzazione. In [10], si propone un "ciclo di vita" per le particelle, che possono muoversi secondo lo schema della PSO, seguire le modalità di selezione, ricombinazione

e mutazione tipiche degli algoritmi genetici, oppure spostarsi secondo l'hill-climbing, con ogni particella che cambia fase dopo che un certo numero di iterazioni non porta un miglioramento.
La PSO è stata usata con successo anche in problemi di ottimizzazione intera: in [11] si nota come sia possibile ottenere ottimi risultati semplicemente troncando, ad ogni iterazione, la parte decimale della posizione delle particelle.

Anche problemi vincolati stati efficacemente affrontati con PSO: un primo approccio prevede di eseguire l'algoritmo classico e di considerare solamente le posizioni valide in fase di aggiornamento delle posizioni migliori. Una seconda tecnica prevede invece l'uso di funzioni di penalità sui vincoli violati. Per funzione di penalità (penalty function) si intende un termine della funzione obiettivo con il compito di penalizzare una soluzione che viola un dato vincolo; essa aumenta il valore della funzione di costo (o diminuisce il valore della funzione di utilità) quando il vincolo è violato, e vale 0 quando quest'ultimo è invece soddisfatto.

La Particle Swarm Optimization è applicata anche a problemi multiobiettivo, si basa generalmente sul mantenimento di un archivio delle posizioni migliori (elitismo), e le diverse implementazioni differiscono principalmente per la strategia di assegnamento della posizione migliore in fase di aggiornamento della velocità di una particella. Una review è presente in [12]: ogni particella può essere influenzata da un diverso ottimo paretiano, scelto, tra gli altri metodi, casualmente o in base a criteri di densità, e in base sia alla popolazione corrente che all'archivio delle soluzioni non dominate. Le posizioni non dominate attraversate possono quindi essere inserite nell'archivio e restituite alla fine.

Implementazione di alcuni algoritmi

In allegato alla presente relazione, è presente un piccolo framework per l'implementazione di algoritmi metaeuristici. Tra i file allegati, `optimizers.h` implementa le funzionalità di base e alcuni algoritmi (tra cui pattern search, una versione di hill climbing per problemi interi, PSO e SPEA2). Il file `main.cpp` mostra esempi di utilizzo degli algoritmi. La documentazione in HTML del framework è presente nella cartella `docs`.

Caratteristiche dell'implementazione

Si è scelto di implementare il framework allegato in C++ per motivi sia di performance che di semplice preferenza personale. Il framework si propone come scopo principale quello di accompagnare la presente relazione fornendo un riscontro pratico a quanto descritto finora.
L'insieme di funzionalità proposte è molto basilare, pensato solamente per fornire delle fondamenta su cui costruire le implementazioni degli algoritmi. Nello specifico, sono state realizate semplici classi per parametri (classe `parameter`, che unifica parametri interi e continui) e problemi di ottimizzazione, più utilità per rendere più snello il codice applicativo.

Si è scelto di implementare l'intera libreria come un solo file header per diverse ragioni. In primo luogo vi è la semplicità per l'utilizzatore del framework di integrarlo in un proprio progetto: non essendo state utilizzate librerie esterne, ed essendo anzi l'intero codice scritto in C++11 standard, è sufficiente includere l'header per poter fare uso del framework. Un approccio alternativo vedrebbe la distribuzione del progetto tramite header puramente dichiarativo più un file di libreria precompilato. Un approccio di questo tipo permetterebbe di nascondere i dettagli implementativi, ed è infatti seguto da risolutori commerciali (ad esempio Gurobi [1] e CPLEX [2]). Al contrario, gli aspetti di didatticità e dimostratività di questo progetto portano a voler esporre i dettagli delle implementazioni piuttosto che nasconderli, anche nella prospettiva di rendere il sorgente disponibile pubblicamente (diventa anche molto semplice in questo caso implementare nuovi algoritmi prendendo ad esempio quelli già implementati, o sperimentare modificando questi ultimi).
Un ultimo approccio consiste nel suddividere l'implementazione in più file, tra header dichiarativi `.h` e sorgenti implementativi `.cpp`, per rendere più gestibile la complessità del framework. Questo approccio è stato evitato principalmente per i motivi di semplicità di integrazione descritti precedentemente, ed è adottato da diversi progetti anche di complessità non banale, come JSON for Modern C++ [3] [4].

[1] `http://www.gurobi.com`
[2] `https://www.ibm.com/analytics/cplex-optimizer`
[3] `https://nlohmann.github.io/json/`
[4] `https://github.com/nothings/stb`

Esempi proposti

Sono proposti esempi per mostrare l'uso e le capacità del framework.

- Problema di ottimizzazione banale, presentato solo allo scopo di illustrare il funzionamento del framework. Per questo motivo, ogni passaggio è maggiormente commentato rispetto agli esempi successivi;
- Riduzione di n-queens a problema di ottimizzazione. I vincoli sono espressi tramite penalty function, secondo l'euristica della minimizzazione del numero di attacchi presentata a lezione. Una penalità è assegnata a valori per le variabili non interi. Per un valore molto piccolo di n (es. 4 o 5) la PSO canonica (basata su variabili a valori reali) risolve velocemente e consistentemente il problema, ma per valori maggiori si rivela quasi del tutto inutilizzabile. Spunti di miglioramento includono l'inclusione delle tecniche per PSO su valori interi, come il passo di approssimazione presentato precedentemente, anche se il problema è tipicamente affrontato con algoritmi specifici per CSP. Successivamente viene proposta una procedura di risoluzione tramite backtracking, che scala ovviamente molto meglio;
- Ottimizzazione di funzione di parametri a valori reali non convessa, caso d'uso tipico di PSO;
- Ottimizzazione di funzione di parametri a valori reali vincolata (funzione di Simionescu);
- Problema multiobiettivo, affrontato tramite SPEA2;
- Nella seconda relazione allegata si mostra un esempio semplice ma concreto di utilizzo del framework, attraverso un caso di studio in cui la PSO si è rivelata efficace. I dettagli sono nel file `rel2.pdf`.

Analizzando gli output ottenuti dal framework sui problemi proposti, si nota come PSO canonica funzioni molto bene come algoritmo generico di ottimizzazione, nonostante le ovvie difficoltà in casi di ottimizzazione di problemi vincolati (es. Simionescu).

Bibliografia e riferimenti

[1] Boyd, Vadenberghe - *Convex Optimization* - 2004

[2] Sörensen, Glover - *Metaheuristics* - 2003

[3] Zitzler, Laumanns, Thiele - *SPEA2: Improving the Strength Pareto Evolutionary Algorithm* - 2001

[4] Sörensen - *Metaheuristics - the metaphor exposed* - 2015

[5] Kennedy, Eberhant - *Particle Swarm Optimization* - 1995

[6] Cheng, Lu, Lei, Shi - *A quarter century of particle swarm optimization* - 2018

[7] Mendes, Kennedy, Neves - *The fully informed particle swarm: simpler, maybe better* - 2004

[8] Kennedy, Mendes - *Neighborhood topologies in fully-informed and best-of-neighborhood particle swarms* - 2003

[9] Zhan, Zhang, Li, Chung - *Adaptive Particle Swarm Optimization* - 2009

[10] Krink, Løvbjerg - *The LifeCycle Model: Combining Particle Swarm Optimisation, Genetic Algorithms and HillClimbers* - 2002

[11] Laskari, Parsopoulos, Vrahatis - *Particle swarm optimization for integer programming* - 2002

[12] Hu, G. Yen - *Adaptive Multiobjective Particle Swarm Optimization Based on Parallel Cell Coordinate System* - 2015