

Particle Swarm Optimization multi-target

Progetto realizzato per il corso di Apprendimento automatico

Docente: prof. Stefano Cagnoni



dott. Longari Michele

Ingegneria informatica spec.

longari@ce.unipr.it

Indice

1. Introduzione	3
<i>a. PSO: Particle Swarm Optimization</i>	3
<i>b. Segmentazione vs. ottimizzazione</i>	4
<i>c. La funzione di fitness</i>	5
<i>d. Le fasi ricerca: globale e locale</i>	6
2. Realizzazione	7
<i>a. Modifiche apportate all'algoritmo originale</i>	8
<i>b. Risultati ottenuti</i>	12
<i>c. Possibili miglioramenti e modifiche future</i>	14
3. Conclusioni	15

1. Introduzione

Nel corso degli anni, la *Particle Swarm Optimization* (PSO) ha destato una sempre maggior attenzione, divenendo oggetto di un crescente numero di ricerche. Il motivo di tale successo è da ricercarsi, oltre che nella semplicità del concetto che ne è alla base, nella versatilità mostrata e nell'efficacia esibita nel risolvere differenti problemi. Il campo applicativo di questo progetto va inquadrato proprio nel tentativo di estendere l'applicazione della PSO all'analisi di immagini, per la ricerca di obiettivi e *target* particolari.

Questo tipo di lavoro è stato compiuto *in primis* col progetto di tesi dell'ing. J. Sartori ("*Particle Swarm Optimization for object detection and segmentation*"). Il progetto in questione è partito da uno studio globale della PSO al fine di individuare gli obiettivi più svariati in un'immagine, per poi focalizzarsi nell'ambito della Visione artificiale sul riconoscimento di targhe automobilistiche.

Il progetto originario dell'ing. J. Sartori consisteva nell'esaminare un'immagine contenente un'autovettura, al fine di estrarne la zona in cui era presente la targa. Con questo progetto invece, realizzato per il corso di Apprendimento automatico, si sono volute estendere le funzionalità del precedente lavoro, per far sì che l'algoritmo riuscisse a concentrarsi e ad estrarre più di un obiettivo-targa in immagini dove fossero presenti più veicoli, anche in condizioni ed angolazioni non ottimali.

Prima di addentrarci nel vero e proprio lavoro che è stato fatto sul progetto, è doveroso prima descrivere a grandi linee come funziona l'algoritmo nella sua versione originaria, sia a livello teorico che esecutivo, per poi così riuscire ad approfondire meglio i punti su cui si è dovuto lavorare maggiormente ed effettuare le modifiche più importanti.

Qualora qualche punto risultasse poco chiaro, o se si desiderassero maggiori approfondimenti, rimandiamo a visionare la relazione completa fatta dall'ing. J. Sartori con la supervisione del prof. S. Cagnoni per la sua tesi di laurea.

a. PSO: Particle Swarm Optimization

L'algoritmo di *Particle Swarm Optimization* (PSO), dal momento della sua pubblicazione nel 1995, è stato impiegato per risolvere numerosi problemi di ottimizzazione. La semplicità del concetto che ne è alla base, unita alla capacità di adattarsi con successo ai vari problemi a cui veniva applicato, ha attirato nel corso degli anni una sempre crescente attenzione. Le origini del PSO affondano le radici nello studio dei comportamenti sociali che regolano i movimenti e le dinamiche di gruppi di animali (stormi di uccelli, banchi di pesci, sciame di insetti, ecc...).

L'aspetto più interessante di questi studi va ricercato nel fatto che l'insieme di individui si muove e si organizza in assenza di un controllo centralizzato. In altre parole, il comportamento collettivo è il

risultato di semplici interazioni locali e il singolo individuo non è dotato di percezione globale. Questo fa sì che, anche con un numero non troppo elevato di particelle, si riescano ad esaminare dati di grandi dimensioni, per poi concentrarsi su una zona specifica, dove una o più delle particelle hanno trovato un possibile *target* secondo le specifiche particolari della funzione di *fitness*.

Ecco nello specifico, come funziona, in pseudo-linguaggio, l'algoritmo di PSO:

```
For each agent
  Initialize agent
End

Do
  For each agent
    Calculate fitness value
    If the fitness value is better than its personal best
      set current value as the new pBest
    End
  End

  Choose the agent with the best fitness value of all as gBest

  For each agent
    Calculate agent velocity
    Update agent position
  End

End

While maximum iterations or minimum error criteria is not attained
```

Da questa semplice descrizione di come opera l'algoritmo, è semplice capire perché a questo tipo di ottimizzazione sia stato dato l'appellativo di "swarm" (propriamente in italiano "sciame"). Tale parola è stata scelta in virtù del comportamento tenuto dai vari elementi della simulazione, il cui movimento caotico sembra molto simile a quello di uno sciame di insetti in cerca di fonti di sostentamento.

Chiarita la correttezza dell'appellativo *Swarm Optimization*, resta ancora da spiegare perché sia stato ad essi anteposto il termine *Particle*. Gli agenti sono degli oggetti senza volume (possono occupare la stessa posizione spaziale) e, almeno nella versione originale, senza massa: questi due concetti porterebbero a classificare gli agenti come dei punti e non come delle particelle. Gli agenti utilizzati sono però anche caratterizzati dall'aver una propria velocità ed accelerazione, due attributi questi strettamente legati al concetto di particella. Per tale motivo gli autori scelsero di battezzare il proprio algoritmo come *Particle Swarm Optimization*.

b. Segmentazione vs. ottimizzazione

In questo paragrafo ci occuperemo di descrivere brevemente come sia stato possibile accomunare i problemi di segmentazione di immagini alla ricerca di particolari obiettivi, con quelli di ottimizzazione; in particolare vedremo come sia stato possibile utilizzare l'algoritmo di PSO per

assolvere questo compito che da principio non sembrerebbe essere tra le sue possibilità applicative. Infatti, sebbene un problema di ottimizzazione (per cui la PSO è stata inizialmente prevista) sia apparentemente diverso da un problema di segmentazione, è possibile trovare un'interessante analogia fra essi. Quando infatti si procede a segmentare un'immagine si vanno a ricercare aree che presentano particolari caratteristiche: ognuna di queste zone può essere vista come l'obiettivo di una specifica ricerca e quindi come l'ottimo di una particolare funzione. Basta allora realizzare una funzione di *fitness* strettamente dipendente dalle caratteristiche cercate affinché si ottenga una convergenza dello sciame nella zona di interesse. Ovviamente vi sono vari problemi aggiuntivi di cui tenere conto, come la molteplicità delle zone da individuare (come ad esempio nel nostro caso dove vi sono più targhe da trovare), la forma non puntiforme dell'obiettivo della ricerca (l'area di interesse è formata da un vasto numero di punti), la presenza di disturbi (punti isolati che casualmente generano alti valori di *fitness*) e altri problemi caratteristici dell'elaborazione di immagini.

Nella segmentazione di immagini al fine di rilevare la presenza di targhe automobilistiche, il primo passo consiste nel convertire l'immagine di partenza (a colori) in una a scala di grigio e calcolarne quindi il gradiente orizzontale. La zona contenente la targa, a causa dell'alternanza fra il nero dei simboli ed il bianco dello sfondo, presenta alti valori di gradiente orizzontale e viene pertanto messa in risalto da quest'operazione. L'immagine gradiente infine viene poi binarizzata mediante un'opportuna soglia, ottenendo così due categorie diverse di pixel: quelli caratterizzati da un gradiente superiore alla soglia, che risultano accesi, e gli altri, che appaiono spenti. La fase successiva consiste nel calcolare il numero di pixel accesi in ogni riga e valutare poi la differenza fra il valore ottenuto per una fila e quello ricavato per la successiva: in corrispondenza dei due picchi più alti vengono fatte coincidere le zone di inizio e fine del posizionamento verticale della targa. In tal modo si riduce sensibilmente lo spazio di ricerca, passando dall'intera immagine ad una striscia. Una volta ottenute così le informazioni sulla dimensione verticale dell'obiettivo, è finalmente possibile calcolarne l'estensione orizzontale, sfruttando il fattore di forma caratteristico delle targhe: il rapporto *larghezza:altezza* è infatti uguale a 5:1. In questo modo si è in grado di costruire un *bounding box* della forma e dimensione della regione cercata; calcolando il massimo della funzione di convoluzione fra tale *box* e la striscia precedentemente individuata è possibile conoscere infine l'esatta collocazione orizzontale (e, di conseguenza, globale) della targa.

c. La funzione di fitness

Nello specifico la funzione di *fitness*, che è necessaria per far sì che le particelle riescano a valutare se una zona soddisfa o meno i requisiti richiesti, è composta di varie condizioni e controlli circa le specifiche che devono avere i pixel che compongono la targa di un'autovettura. Principalmente si può dire che per implementare *fitness* è stato utilizzato un controllo sul colore del pixel e l'operatore gradiente, ovvero l'operatore che segnala quanto sia alto il cambiamento di colore tra

un pixel e quelli adiacenti. E' chiaro che nelle targhe, per la continua alternanza tra zone nere (i caratteri che compongono la sigla) e zone bianche (lo sfondo), l'operatore gradiente sarà in grado di restituire dei valori molto alti in corrispondenza della zona interessata per i cambi netti e repentini di colore. L'effetto combinato delle analisi del colore e del gradiente fanno sì che le particelle favoriscano la ricerca dei pixel bianchi (neri) confinanti con altri punti neri (bianchi): questa è la situazione tipica che si verifica in corrispondenza di una targa. Il comportamento atteso sarà quindi quello di una convergenza dello sciame nella zona da noi ricercata. Purtroppo però, nei casi reali, la presenza sulle targhe di ombre e riflessi ha l'effetto di alterare i colori, per questo motivo si rende necessario ampliare lo spettro delle tonalità ammesse, includendo l'intera scala di grigi.

Vediamo quindi, dopo averla descritta a parole, l'effettiva implementazione della funzione di *fitness*:

```

if(|r(x,y)-g(x,y)|>max_diff or |r(x,y)-b(x,y)|>max_diff or |g(x,y)-b(x,y)|>max_diff)
    fitness = 0;
else
{
    gradiente_dx = |grayscale(x,y)-grayscale(x+1,y)|;
    gradiente_sx = |grayscale(x,y)-grayscale(x-1,y)|;

    if(gradiente_dx>gradiente_sx)
        fitness = gradiente_dx ;
    else
        fitness = gradiente_sx ;
}

```

Una funzione di *fitness* come quella definita sopra non è purtroppo sufficiente ad assicurare un buon comportamento dello sciame, che potrebbe essere ingannato da disturbi isolati (pixel, di una tonalità ammessa, ad alto valore di gradiente). Per evitare che lo sciame venga attirato verso un disturbo isolato possiamo introdurre anche questa volta una componente di *fitness* locale, da affiancare a quella puntuale definita nelle pagine precedenti. La *fitness* locale rappresenta una valutazione della bontà della zona: tanto più è elevata, tanto più l'area circostante presenta pixel interessanti (tonalità ammessa e gradiente elevato). La funzione di *fitness* complessiva sarà la somma dei contributi puntuale e globale: $fitness(x, y) = fitness\ puntuale(x, y) + fitness\ locale(x, y)$.

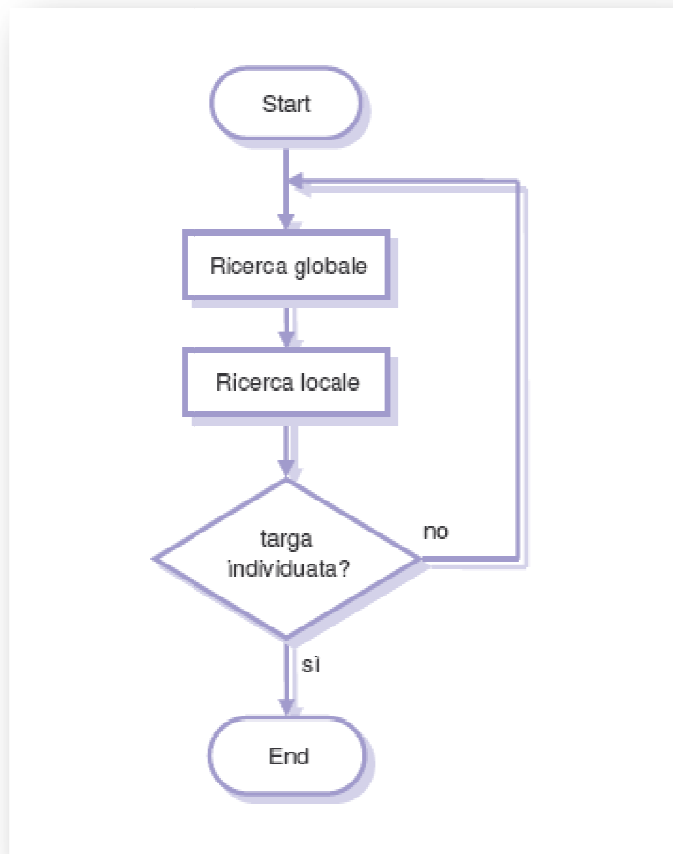
Descritta la *fitness function*, passiamo ora a descrivere la fase topica dell'algorithm, ovvero la ricerca vera e propria della zona cercata.

d. Le fasi di ricerca: globale e locale

Nel paragrafo precedente abbiamo definito le regole tramite cui le particelle valutano la bontà della propria posizione spaziale e si muovono di conseguenza. Ora si tratta di capire come impiegare gli agenti implementati per localizzare l'obiettivo della ricerca: la targa. L'idea di

massima sarà utilizzare una ricerca a due fasi. Nella prima le particelle esploreranno l'intero dominio (l'immagine) con lo scopo di individuare le zone più interessanti, quelle cioè con maggior probabilità di rappresentare la targa. Nella seconda fase l'attenzione sarà focalizzata solo nelle aree appena trovate: gli agenti procederanno alla loro esplorazione in modo più accurato, al fine di determinare se ad una di esse corrisponda realmente la presenza della targa. Se l'operazione produrrà un esito positivo affermeremo di aver localizzato l'obiettivo; in caso contrario occorrerà ripetere da capo le due fasi di ricerca.

Nell'immagine seguente è riassunta la sequenza delle fasi descritte in precedenza:



Nel caso in cui vengano terminate con successo entrambe le fasi di ricerca, si procede con l'individuazione e la successiva estrazione vera e propria della targa. Entrambe le operazioni vengono fatte in base alla disposizione assunta da ciascuna delle particelle del sotto-sciame "esploratore" che ha avuto successo nella sua fase di ricerca dell'obiettivo.

2. Realizzazione

La parte di realizzazione e implementazione di questo progetto, si può dire che è stata composta di due fasi principalmente: una prima fase di studio approfondito dell'algoritmo argomento di lavoro, e una seconda fase di modifiche e aggiornamenti allo stesso.

Prima di tutto c'è da dire che l'impostazione che si è voluta dare a questo progetto, è stata quella di mantenersi il più fedeli possibili all'algoritmo da cui si partiva a lavorare, andando a modificare solo le parti strettamente necessarie, oppure quelle dove erano ben visibili possibili miglioramenti.

Quindi, si può dire che la versione "aggiornata" dell'algoritmo ottenuta alla fine del progetto, è nel complesso molto simile a quella precedente; però sono state aggiunte le funzionalità di gestione di target multipli, di controllo sull'eliminazione di zone dall'immagine (con o senza obiettivi) ed infine è stata implementata l'eliminazione (durante la ricerca locale) di zone "rumorose" in grado di trarre in inganno gli agenti.

Passiamo ora a vedere nel dettaglio i cambiamenti apportati e le modifiche fatte sul codice.

a. Modifiche apportate all'algoritmo originale

Prima di iniziare l'analisi nel suo complesso, c'è da precisare che i cambiamenti che sono stati apportati al progetto originale redatto dall'ing. J. Sartori, sono stati sostanzialmente quasi tutti concentrati nelle due fasi di ricerca, come era anche logico aspettarsi.

Nonostante questo, è stata apportata anche qualche altra modifica ugualmente importante in parametri, costanti e funzioni di I/O.

Ma ora partiamo ad analizzare il lavoro fatto con ordine, seguendo l'iter che compie l'algoritmo, da quando viene lanciato a quando termina.

Prima di tutto partiamo dall'esecutore del programma (file main.c). Una volta era qui che venivano fatti i controlli e restituiti i risultati dell'algoritmo. Adesso invece il ruolo di "gestore" vero e proprio è affidato alla PSO (file pso.h e pso.c), e al *main* rimane il compito solo di lanciare il programma e di chiuderlo correttamente nel caso in cui si sia in assenza di errori. Questo è stato fatto per poter seguire meglio parametri e costanti dell'algoritmo in esecuzione; ma soprattutto perché, introducendo la funzionalità *multi-target*, è stata cambiata la modalità di funzionamento dell'algoritmo nelle sue fondamenta, e questo cambiamento non permetteva più un controllo esterno ad esso e decentralizzato.

Quindi, come si è potuto intuire, il ruolo centrale di gestione dell'algoritmo, è affidato alla PSO stessa. In pratica qui vengono lanciate a ripetizione le due fasi di ricerca; poi di volta in volta, nel caso in cui la ricerca locale abbia trovato un possibile obiettivo, viene eseguita la scrittura dell'output, così da poter vedere chiaramente i risultati. La PSO restituisce il controllo al *main* in due casi: o dopo che si è raggiunto un numero massimo di tentativi (settati a 100), o dopo che si è arrivati ad un numero massimo di successi desiderato (per ora fissato a 5).

Le procedure descritte sopra è possibile vederle meglio in questo stralcio di codice preso appunto dal file pso.c:

```
do
{
  attempt++;

  ... ..

  /** Effettuo la ricerca globale **/
  global_found = global_search(params, agents, img, &a);

  ... ..

  /** Effettuo la ricerca locale **/
  local_found = local_search(params, global_found, a, agents, img, &plate);

  if(local_found)
  {
    successes++;
    if(params->writeOutput) write_output(params, local_found, plate, img);
  }

  /**
   Dopo un certo numero di ricerche senza successo, riassetto la cache.
   Questo cerca di rimediare all'eventuale cancellazione erronea di una targa.
  ***/

  if(attempt== ATTEMPTS_NUM/2 && successes== 0) initCache(cache, img.w, img.h);

} while ( successes < TARGET_NUM && attempt < ATTEMPTS_NUM );
```

Un'altra modifica importante fatta in questo contesto è stata nella funzione *updateFitnessPuntuale()*. In pratica il compito di questa funzione è quello di aggiornare di volta in volta la funzione di *fitness* puntuale di ogni particella. Qui è stato aggiunto un controllo più netto riguardo la *cache* dell'immagine, in quanto erano molteplici i casi di *update* in zone dell'immagine non compatibili col procedere dell'algoritmo, o in zone che erano state eliminate in precedenza negativizzando la *cache* (di questo parleremo bene più avanti quando descriveremo nel dettaglio le funzioni *loc_del_zone()* e *loc_del_targa()*).

Detto ciò possiamo ad analizzare la prima fase di ricerca che viene effettuata che, come è reso chiaro anche dal codice sopra riportato, è la ricerca globale (file *global_search.h* e *global_search.c*). La prima modifica che è stata fatta in questa parte, riguarda immediatamente la funzione *global_initParticles()*. Infatti, anche qui è stato aggiunto un controllo più accurato sulla *cache*, inizializzando le particelle solo dopo aver controllato che la zona in questione non sia una di quelle tolte nelle precedenti iterazioni dell'algoritmo. Ecco il controllo descritto sopra:

```
if( cache[cache_position] >= -1 )

// controllo fatto per evitare di inizializzare particelle
// in zone precedentemente eliminate ( cache a -2 o -3 )

{

agents[a].currentValue = fitness_puntuale(agents[a].x, agents[a].y, img);
agents[a].bestValue = agents[a].currentValue;
global_rep_map[agents[a].x/REP_X + agents[a].y/REP_Y * rep_map_width] += 1;

addToMap( neighborhood_map.map, agents[a].x/NEIGHBOURHOOD_RANGE +
          agents[a].y/NEIGHBOURHOOD_RANGE * neighborhood_map.w, a );

a++;

}
```

Un'altra modifica fatta nella *global search* riguarda la funzione *global_updateFitness()*. Anche qui come nel file *pso.c* è stato aggiunto il controllo sulla *cache*, ed in più è stato tolto il calcolo della *fitness* locale per agenti che si trovano in zone eliminate in precedenza. Questo è stato fatto per evitare che una particella che si trova in quel momento in una zona eliminata, possa inavvertitamente aumentare la sua *fitness*, e così creare problemi di “falso ritrovamento” alle successive iterazioni dell’algoritmo.

Direttamente successiva alla ricerca globale, come abbiamo visto, vi è quella locale, che parte dai risultati precedentemente raggiunti, per poi controllarli ed approfondirli. Nella *local search* (file *local_search.h* e *local_search.c*), a differenza della *global search*, la fase di inizializzazione è stata lasciata uguale per due motivi. Prima di tutto perché, col controllo fatto sulla *global search*, dovrebbero verificarsi molto raramente situazioni di inizializzazione in zone erronee. In secondo luogo perché si era notato che, col procedere dell’algoritmo, si rischiavano situazioni di “stallo”, dato che sempre meno erano le zone a *cache* maggiore o uguale a -1 (cioè le zone con pixel inesplorati) disponibili in una zona locale in analisi. Finita la ricerca, come anticipato nel primo capitolo, si ha l’individuazione della targa (o comunque della possibile zona che la contiene) e la successiva estrazione. Quest’ultima operazione viene fatta tramite la funzione *extract_plate()* (presente nel file *functions.c*). Anche questa funzione è stata leggermente modificata nel finale con un controllo pixel per pixel della zona estratta. Inoltre al termine di questa operazione, per qualsiasi risultato ottenuto, sono state aggiunte due operazioni di cancellazione: una per la zona e una per gli agenti. Questo è stato fatto per far sì che ci fosse una sorta di *reset* completo sugli agenti esploratori e sulle zone da esplorare.

In questo ambito è necessario ora descrivere meglio, per maggiore chiarezza, le funzioni di cancellazione che sono state nuovamente implementate. Esse sono: *loc_del_zone()* e *loc_del_targa()*. Prima ancora però, introduciamo il concetto che è stato usato di *cache* dell’immagine. Per velocizzare le operazioni di valutazione della *fitness* puntuale di un pixel è stata inserita una *cache*: in pratica essa è una matrice di interi, con una cella per ogni pixel, inizializzata con tutti i valori a -1. Ogni volta che si giunge su un pixel si controlla se esso è già stato visitato o

meno. Se il valore è -1, esso non è mai stato valutato e pertanto lo si esplora e si aggiorna la *cache*. Se il valore è maggiore o uguale a 0, quel valore è pari alla sua *fitness* puntuale. Se il valore invece è -2 o -3, significa che quel pixel va evitato, poiché è stato precedentemente eliminato, rispettivamente, dalla *loc_del_zone()* o dalla *loc_del_targa()*. Quindi queste funzioni non vanno ad operare sull'immagine, ma su una sorta di "memoria storica" della stessa. Ecco qui di seguito riportato il semplice codice che definisce queste due funzioni:

```
/** Elimina una zona dell'immagine, forzandola */
/** ad assumere una fitness puntuale negativa. */
void loc_del_zone(int min_x,int max_x,int min_y,int max_y,int w,int* cache)
{
    int i, j;

    for (i = min_x; i <= max_x; i++)
        for (j = min_y; j <= max_y; j++) cache[i+j*w] = -2;
}

/** Elimina una targa dall'immagine, forzandola */
/** ad assumere una fitness puntuale negativa. */
void loc_del_targa(int min_x,int max_x,int min_y,int max_y,int w,int* cache)
{
    int i, j;

    for (i = min_x; i <= max_x; i++)
        for (j = min_y; j <= max_y; j++) cache[i+j*w] = -3;
}
```

Concludendo, si può affermare che, per quanto riguarda il nucleo dell'algoritmo vero e proprio, le modifiche principali possono essere tutte ricondotte a quelle appena elencate e descritte.

Altre modifiche sono state fatte ad alcune costanti e parametri progettuali, per renderli più adatti al tipo di *testing-set* usato. Alcune di queste modifiche riguardano per esempio il *REPULSION_RANGE*, ovvero la distanza entro cui entra in gioco la repulsione; il *NEIGHBOURHOOD_RANGE*, che è la distanza entro cui un punto è considerato "vicinato"; la *MAX_RGB_DIFF*, cioè la costante che rappresenta la massima differenza fra i valori RGB ammessa affinché un pixel sia considerato accettabile durante la fase di valutazione della *fitness* ed infine le costanti di *RATIO* e *DIM* delle targhe, che servono come veri e propri "parametri costruttivi" di riferimento.

Un'ultima cosa da specificare, prima di finire questa descrizione, è che generalmente, nell'algoritmo, sono state scelte condizioni di terminazione e parametri non troppo restrittivi e rigorosi, per far sì che la PSO avesse più facilità nonché probabilità di trovare tutte le targhe presenti, magari rischiando però di sbagliare e segnalarne una che in realtà non è tale.

Questa decisione è stata presa perché, pensando a questo lavoro come il primo passo di un processo più complesso, quale l'estrazione ed il riconoscimento dei caratteri che costituiscono la targa stessa, è evidente che nel caso in cui venisse estratta una "falsa targa", al passo successivo sarebbe sicuramente riconosciuta come tale e quindi scartata. Peggioro, dal punto di vista dei risultati e della funzionalità, sarebbe invece se, una "vera targa", magari perché si trova in

condizioni d'angolazione o illuminazione particolari, non venisse segnalata, e quindi successivamente la si perdesse in maniera definitiva.

b. Risultati ottenuti

In questo paragrafo ci occuperemo dell'analisi dei risultati ottenuti in base al lavoro fatto. I dati saranno principalmente analizzati in termini di tempi di elaborazione e di percentuale di successi.

Per prima cosa precisiamo che i risultati non possono essere paragonati con quelli dell'algoritmo a singolo obiettivo per vari motivi. Prima di tutto per la differente implementazione che è stata fatta, che prevede più iterazioni, più particelle e soprattutto che cerca ripetutamente più *target*. Poi non è nemmeno da tralasciare il fatto che per il *testing-set* sono state usate immagini con una risoluzione più che doppia rispetto alle precedenti. Infatti si sono adoperate per i test 15 immagini in formato PPM, con risoluzione 640x512. In queste immagini erano presenti da un minimo di una ad una massimo di cinque automobili con le rispettive targhe, fotografate in diverse condizioni di luce, angolazione e locazione. In totale il *testing-set* completo comprendeva 31 targhe. Gli aspetti che andremo ad analizzare sono principalmente rappresentati dal numero di targhe correttamente individuate e dalle prestazioni temporali. Và segnalato che la macchina su cui sono stati effettuati i test è equipaggiata con una CPU INTEL PENTIUM M a 1.8 GHz e 1,5 GB di RAM. Inoltre, per avere risultati il più rappresentativi possibili, sono state fatte 3 serie da 10 prove sul *set* di immagini a disposizione. Ognuna delle tre serie variava per la rigorosità dei parametri e delle condizioni di terminazione. In questo modo è stato anche possibile notare come evolvesse il rapporto prestazioni/tempo, mano a mano che si procedeva.

Ecco di seguito riportata la prima tabella, quella che riporta i dati ottenuti, settando i valori più restrittivi possibili per il numero di iterazioni e per le condizioni di uscita con successo:

<i>Numero prova</i>	<i>Numero individuate corrette</i>	<i>Numero individuate errate</i>	<i>Numero non individuate</i>	<i>Tempo medio di elaborazione</i>
1	29	8	2	38,35
2	28	6	3	43,21
3	30	6	1	41,06
4	27	8	4	34,21
5	28	7	3	37,14
6	30	8	1	43,26
7	28	6	3	33,78
8	29	8	4	37,14
9	29	7	2	41,32
10	30	6	3	35,07

E' evidente che si hanno degli ottimi risultati in termini di successi (circa il **90%** delle targhe è riconosciuto correttamente ad ogni iterazione), ma pessimi in termini di tempi di elaborazione (in media per ogni immagine si impiegano **38,5 s**). Inoltre si nota che il numero di "falsi obiettivi" individuati (intorno ai **7** per iterazione), è parecchio elevato.

Qui di seguito è riportata la seconda tabella, che rappresenta anche la seconda delle tre serie di *testing*. I parametri sono sistemati, per cercare di abbassare i tempi di elaborazione:

Numero prova	Numero individuate corrette	Numero individuate errate	Numero non individuate	Tempo medio di elaborazione
1	24	4	11	2,93
2	30	4	5	3,21
3	28	4	7	3,07
4	26	5	9	2,97
5	25	5	10	3,11
6	31	3	4	3,27
7	28	3	7	3,26
8	27	5	8	3,32
9	30	5	5	3,23
10	24	4	11	3,15

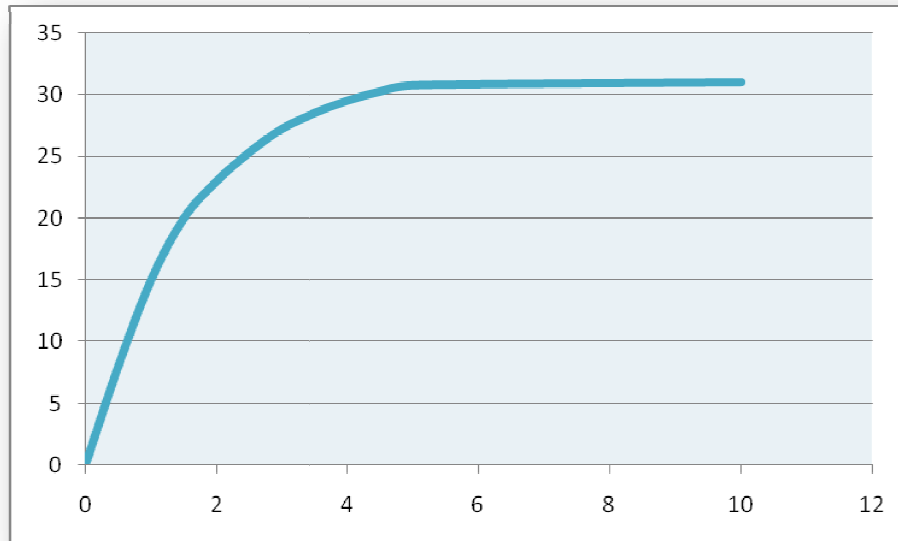
Come si può vedere, i tempi si sono abbassati di circa dieci volte (ora si impiegano in media **3,15 s**), mentre i successi non sono diminuiti così drasticamente (siamo sull'**85%**). Questo testimonia che non è necessario impostare condizioni troppo restrittive per avere dei buoni risultati.

Vediamo ora la terza ed ultima serie di prove che risultati ci ha dato:

Numero prova	Numero individuate corrette	Numero individuate errate	Numero non individuate	Tempo medio di elaborazione
1	20	1	15	1,12
2	18	0	17	0,93
3	21	1	14	1,03
4	19	1	16	0,86
5	17	2	18	1,43
6	20	1	15	0,97
7	18	1	17	1,18
8	21	2	14	0,84
9	17	0	18	1,09
10	20	0	15	1,19

Si nota molto bene che qui il *setting* dei parametri è stato scelto esageratamente “permissivo”, in quanto i tempi si sono abbassati nuovamente (in media siamo solo su **1 s** per elaborare un’immagine), però di pari passo sono peggiorate tantissimo le prestazioni (si ha solo il **59%** di successi).

Per concludere la fase di analisi/test, è stato studiato, col seguente grafico, l’andamento del numero di targhe correttamente classificate in funzione del tempo:



Dopo queste accurate prove, e dopo un’attenta analisi del grafico prestazioni/tempo, sono stati scelti come parametri di *default* per il progetto (in particolare per *ATTEMPT_NUM* e per le due condizioni di terminazione della *global search* e della *local search*) valori che si pongono a metà via tra quelli della prima e della seconda serie. Si è così ottenuto un tempo medio di elaborazione e una percentuale di successi che si avvicinano alle due migliori configurazioni (il tempo medio è fissato sui **4,23 s** in media, invece la percentuale di successi è circa del **93,54%**).

c. Possibili miglioramenti e modifiche future

Nonostante questo progetto partisse da solide fondamenta, e nonostante il lavoro effettuato per aggiornare e potenziare l’algoritmo, proprio per le caratteristiche di portabilità ed elasticità della PSO, durante il lavoro sono state notate numerose modifiche e miglioramenti da poter fare con futuri progetti da accorpate a quelli già terminati.

In primo luogo si potrebbe lavorare per definire in maniera più rigorosa come debbano essere le immagini da analizzare. Bisogna prima di tutto decidere una risoluzione delle immagini univocamente accettabile in termini di tempi e di qualità, ma anche impostare una sorta di

“standard” che definisca distanza, altezza e angolazione da cui vengono scattate le fotografie. Così facendo si riuscirebbe a lavorare meglio sui parametri progettuali e soprattutto a prendere decisioni “ad hoc” altrimenti difficili se non impossibili da poter prendere (come per esempio il numero di iterazioni, la dimensione approssimativa in pixel delle targhe, il numero di agenti, i limiti di accettabilità del rapporto *altezza:larghezza*, i parametri di repulsione e vicinato, ecc...). Queste definizioni di canoni univoci vanno attuate cercando di mediare tra la necessità di avere immagini poco “rumorose” e quella di poter analizzare anche immagini in cui le targhe appaiono in secondo piano.

In secondo luogo un aspetto su cui lavorare è sicuramente quello del perfezionamento dell’algoritmo in tutte le sue fasi, per far sì che si ottengano risultati sempre migliori come tempi e soprattutto come percentuali di successi. Tra questi perfezionamenti ci potrebbe essere anche il sistemare l’implementazione del *tracking* (già presente nel progetto dell’ing. J. Sartori) in modalità compatibile con i *multi-target*. Anche se c’è da precisare che la difficoltà di eseguire un *tracking* con più obiettivi è piuttosto marcata, e probabilmente non molto utile ai fini delle applicazioni per cui può essere usato materialmente l’algoritmo di PSO *multi-target*.

Infine, un ultimo punto su cui lavorare è risolvere un comportamento indesiderato che si è riscontrato più volte durante le fasi di *testing*. In pratica, quello che capitava, era che in un’immagine del testing-set, dove erano presenti due targhe in primo piano e tre sullo sfondo, quest’ultime venissero spesso tralasciate o non correttamente elaborate dall’algoritmo, a causa del numero elevato di particelle che attiravano a sé i due *target* più grandi in primo piano. Varie sono le possibili soluzioni a questo problema, ma la strada che sembra più promettente è quella di settare una sorta di termine di “guadagno” che vada via via diminuendo nel procedere dell’algoritmo. Questo termine potrebbe guidare le particelle a cercare (anche forzatamente) obiettivi di dimensioni più piccole, qualora l’algoritmo si concentri troppo su *target* di grandi dimensioni già segnalati. Un’altra possibile idea, è quella di controllare i valori delle costanti di NEIGHBOURHOOD_RANGE e REPULSION_RANGE, e fare in modo che, anche con un eventuale cambiamento delle dimensioni delle targhe, non nascano problemi di compatibilità.

3. Conclusioni

Tramite questo lavoro si è approfondito un campo di recente applicazione per la *Particle Swarm Optimization*: l’elaborazione di immagini, volta all’individuazione di particolari oggetti contenuti al loro interno.

I commenti e gli approfondimenti che si possono fare su questo tipo di lavoro e sulle sue applicazioni, sono molti, anche perché come argomenti di ricerca sono molto odierni. In modo particolare le possibili applicazioni di questo algoritmo sono molteplici e di pubblica utilità. Infatti, il problema dell’individuazione della collocazione spaziale di più targhe in un’immagine può essere visto come il primo passo di un processo più complesso, quale l’estrazione ed il riconoscimento dei

caratteri che costituiscono la targa stessa. In tal modo si può creare un sistema automatizzato in grado di sorvegliare il traffico veicolare e le zone di sosta delle automobili, infliggere multe, controllare gli accessi alle zone a traffico limitato (ZTL) e addirittura ricercare automobili rubate fra quelle in transito o in sosta nei parcheggi, magari con un'analisi sezione per sezione degli stessi.

E' chiaro che per questo tipo di applicazioni, le *performance* ottenute (anche se migliorabili), sono più che accettabili. Invece se si volesse applicare l'algoritmo per assolvere a compiti propri della Visione artificiale, sarebbe più proprio l'utilizzo dell'algoritmo *mono-target* con la funzionalità di *tracking* abilitata, che ha presentato tempi di elaborazione e qualità molto soddisfacenti.

Concludendo, si può affermare che si è notato come l'algoritmo nuovamente modificato proposto in questo progetto presenti buone capacità di individuazione degli obiettivi, ma anche che, per ottenere una miglior definizione delle zone perimetrali degli oggetti ricercati, è opportuno affiancargli metodi tradizionali di analisi delle immagini. I risultati ottenuti sono decisamente accettabili per certi livelli applicativi, ma saranno comunque necessari ulteriori test su altri problemi di elaborazione di immagini e di filmati, per verificare quanto effettivamente la soluzione proposta sia flessibile e generale.

Fra i possibili sviluppi della ricerca, un aspetto interessante potrebbe essere rappresentato dallo studio di modelli alternativi di organizzazione degli sciame (come ad esempio mediante l'utilizzo del *clustering k-mean*, un algoritmo che sembra decisamente adatto a questo scopo) e dall'analisi di come questi influenzino le prestazioni esibite.