UNIVERSITÀ DEGLI STUDI DI PISA
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Scienze dell'Informazione

Tesi di Laurea

Algoritmi genetici e simulated annealing per la soluzione parallela di problemi di ottimizzazione combinatoriale

Candidati

Marco Bucci

Donato Circelli

Relatori

Dott. Raffaele Perego

Dott. Ranieri Baraglia

Controrelatore

Dott.ssa M.G. Scutellà

Anno Accademico 1995/96

Ringraziamenti

Ringraziamo tutti coloro che hanno contribuito al nostro lavoro.
Un riconoscimento particolare ai nostri relatori, Dott. Raffaele Perego e Dott. Ranieri Baraglia, per la loro continua disponibilità, competenza e professionalità.
Vogliamo inoltre ringraziare tutto l'Istituto CNUCE (Centro Nazionale Universitario di Calcolo Elettronico) del CNR (Centro Nazionale delle Ricerche) sito in Pisa, via S. Maria n.36, per le innumerevoli risorse tecniche che ci sono state messe a disposizione.
Un grazie particolare, inoltre, agli amici conosciuti durante la nostra assidua presenza nel centro di calcolo parallelo del CNUCE. La loro simpatia e disponibilità ha reso meno pesante il nostro compito durante l'anno di lavoro. Grazie dunque a Michele, Giovanni, Marco e Massimiliano.
Un ulteriore ringraziamento al Prof. Rocco De Nicola del Dipartimento di Sistemi e Informatica della Facoltà di Ingegneria di Firenze per le risorse messe a disposizione nelle ultime due settimane di lavoro.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

In questa tesi vengono studiati alcuni algoritmi euristici per la ricerca delle soluzioni di problemi di ottimizzazione combinatoriale. È noto che la maggior parte di questi problemi è intrattabile non appena la dimensione del problema diviene significativa. Il ricorso ad algoritmi di approssimazione con complessità polinomiale è quindi indispensabile quando la dimensione dei problemi cresce. Anche gli algoritmi euristici sono tuttavia caratterizzati da complessità computazionali elevate che rendono spesso difficile trovare buone soluzioni in tempi ragionevoli. Per ovviare a questa limitazione, che spesso costringe a sacrificare la qualità della soluzione a scapito di tempi di esecuzione accettabili, è comune nella comunità scientifica il ricorso al parallelismo.

Il lavoro di tesi riguarda lo studio, il progetto e l'implementazione su una macchina ad elevato parallelismo, un *multicomputer* nCUBE2 con 128 nodi di elaborazione, di *algoritmi genetici* e dell'algoritmo di *Simulated Annealing*. Il problema usato come caso di studio è il classico problema del commesso viaggiatore (TSP), che per la sua semplicità formale ben si presta ad essere usato come *benchmark* per la valutazione di euristiche.

Entrambi gli algoritmi trattati si basano su un'analogia con processi che avvengono realmente in natura: la riproduzione delle specie viventi sessuate, e il processo di raffreddamento dei solidi.

Gli *algoritmi genetici* sono algoritmi stocastici nei quali la ricerca delle soluzioni è condotta imitando il comportamento degli organismi viventi che hanno una riproduzione sessuata; si generano delle *popolazioni* di soluzioni e si selezionano i migliori individui, analogamente a quanto avviene in natura secondo la teoria evolutiva di Darwin. Inoltre vengono sfruttati i concetti di ereditarietà e sopravvivenza dell'individuo più adatto. In natura, per la maggior parte degli organismi viventi, l'evoluzione avviene attraverso due processi fondamentali: la selezione naturale e la riproduzione sessuale. La prima determina quali elementi di una popolazione sopravvivono per riprodursi, la seconda garantisce il rimescolamento e la ricombinazione dei geni dei loro discendenti. Quando spermatozoo e uovo si uniscono, i cromosomi corrispondenti si allineano, poi si incrociano e si separano scambiandosi materiale genetico. Questo rimescolamento consente una evoluzione molto più rapida di quella che si avrebbe se tutti i discendenti contenessero semplicemente una copia dei geni di un unico genitore, modificata di tanto in tanto per mutazione.

Le soluzioni trattate da un *algoritmo genetico* vengono dunque combinate tra loro attraverso alcuni operatori genetici, che operano sulla popolazione di soluzioni generandone continuamente di nuove in un ciclo simile a quello delle specie viventi sessuate.

Le soluzioni che mostrano di essere più adatte delle altre, sopravvivono alla prossima generazione e si riproducono di nuovo. Le soluzioni ritenute peggiori vengono scartate, e non hanno quindi discendenza. In questo modo si raffinano le soluzioni, scartando quelle meno buone e permettendo solamente a quelle migliori di riprodursi. Il processo ha termine quando gli operatori genetici non riescono a migliorare ulteriormente le soluzioni trovate.

Gli algoritmi genetici hanno mostrato di non essere in generale capaci di convergere verso un ottimo globale se non si includono al loro interno procedure di ricerca locale (vedi capitolo 3); in effetti la caratteristica degli algoritmi genetici non sta tanto nel fatto che essi trovano soluzioni ottime globali, ma nella loro capacità di produrre molte soluzioni buone. Come nella realtà, l'evoluzione migliora la specie in generale, ma individui con caratteristiche speciali sono alquanto rari.

Nella sua forma originaria [27] l'algoritmo di *Simulated Annealing* è basato sull'analogia tra il processo di solidificazione dei solidi e il problema della risoluzione di problemi combinatori complessi. In fisica, il termine **annealing** denota un processo fisico nel quale un solido viene riscaldato progressivamente fino ad una temperatura alla quale raggiunge lo stato liquido, per poi farlo raffreddare lentamente. In questo modo, se la temperatura raggiunta è abbastanza alta e il processo di raffredddamento sufficientemente lento, tutte le molecole si dispongono in una configurazione di minima energia.

Per analogia con il processo fisico di *annealing*, l'algoritmo di *Simulated Annealing* incorpora il concetto di temperatura; quest'ultima, inizialmente alta, viene abbassata progressivamente nel corso della ricerca delle soluzioni. Per ogni temperatura si effettuano un numero prestabilito di iterazioni dell'algoritmo.

Il concetto di temperatura entra in gioco per quanto riguarda il criterio di accettazione o rifiuto di nuove soluzioni, che è la particolarità più interessante dell'algoritmo di *Simulated Annealing*. Infatti finchè la temperatura è alta, il criterio di accettazione delle nuove soluzioni prevede che si possa accettare anche una nuova soluzione che risulti meno buona di quella precedente. In questo modo, nella fase iniziale della ricerca, si riescono a superare gli ottimi locali, e la ricerca di soluzioni buone si distribuisce quindi con maggiore uniformità nello spazio delle soluzioni. Infine, quando la temperatura è sufficientemente bassa, vengono accettate esclusivamente soluzioni migliori di quelle correntemente trattate. L'algoritmo di *Simulated Annealing* coincide quindi, per temperature basse, con un algoritmo di ricerca locale.

Il lavoro di tesi è strutturato nel seguente modo:

- nel capitolo 1 sono presentati gli *algoritmi genetici*, gli operatori genetici che operano sulla popolazione di soluzioni, ed è riportata una breve esposizione della teoria su cui si basano detti algoritmi. Inoltre viene presentato un risultato fondamentale, noto come *Teorema degli schemi*;

- nel capitolo 2 sono presentate varie metodologie di implementazione degli *algoritmi genetici* su elaboratori paralleli, e ne vengono esaminate caratteristiche e particolarità;

- nel capitolo 3 è riportata una descrizione dell'algoritmo di *Simulated Annealing*, ed è presentato un algoritmo *ibrido* che combina l'algoritmo di *Simulated Annealing* con una procedura di ricerca locale;

- nel capitolo 4 vengono riportate alcune soluzioni al problema di implementare su elaboratori paralleli l'algoritmo di *Simulated Annealing*;

- nel capitolo 5 sono presentati vari algoritmi genetici sequenziali e paralleli esistenti in letteratura applicati al problema del commesso viaggiatore, implementati durante il lavoro di tesi su un'architettura NCube2. Alcune di queste implementazioni adottano strategie originali. In particolare è stato sviluppato un *algoritmo genetico* a *grana fine* in cui la dimensione della popolazione non dipende dal numero di nodi usati per eseguire l'algoritmo, limitazione questa che riguarda la maggior parte delle implementazioni proposte in letteratura. I risultati delle prove condotte su questi algoritmi hanno permesso di valutare il funzionamento di vari tipi di operatori genetici e di confrontare le prestazioni degli *algoritmi genetici* con l'algoritmo di *Simulated Annealing*;

- nel capitolo 6 viene presentato un algoritmo *ibrido* sviluppato durante il lavoro di tesi. L'algoritmo, denominato AGSA dalle iniziali dei termini *Algoritmi Genetici* e *Simulated Annealing*, combina i due algoritmi cercando di sfruttare le principali caratteristiche positive di entrambi gli algoritmi originari.

Seguono le Conclusioni, le Appendici e la Bibliografia.

Capitolo 1

Algoritmi genetici

Gli algoritmi genetici appartengono alla classe degli **algoritmi di approssimazione** (o **euristici**) usati per risolvere svariati problemi **NP**[1], tra cui anche quelli di ottimizzazione combinatoriale [33].

Gli algoritmi genetici sono algoritmi stocastici nei quali la ricerca delle soluzioni è condotta imitando il comportamento degli organismi viventi che hanno una riproduzione sessuata; si generano delle **popolazioni** di soluzioni e si selezionano i migliori individui, analogamente a quanto avviene in natura secondo la teoria evolutiva di Darwin. Inoltre vengono sfruttati i concetti di ereditarietà e sopravvivenza dell'individuo più adatto. Infatti, in natura, per la maggior parte degli organismi viventi, l'evoluzione avviene attraverso due processi fondamentali: la selezione naturale e la riproduzione sessuale. La prima determina quali elementi di una popolazione sopravvivono per riprodursi, la seconda garantisce il rimescolamento e la ricombinazione dei geni dei loro discendenti. Quando spermatozoo e uovo si uniscono, i cromosomi corrispondenti si allineano, poi si incrociano e si separano scambiandosi materiale genetico. Questo rimescolamento consente una evoluzione molto più rapida di quella che si avrebbe se tutti i discendenti contenessero semplice-

[1] Si dicono **NP** quei problemi decisionali risolubili in tempo polinomiale da un algoritmo non deterministico.

mente una copia dei geni di un unico genitore, modificata di tanto in tanto per mutazione.

Per poter impiegare gli algoritmi genetici (nel seguito useremo l'abbreviazione AG) è necessario innanzitutto stabilire una codifica della soluzione del problema trattato in termini di stringhe binarie (dalla definizione originaria di Holland che vedremo nel prossimo Paragrafo); la ricerca di una buona soluzione diventa quindi la ricerca di particolari stringhe binarie [38].

Si parte da una popolazione iniziale di **individui**, ciascuno dei quali mantiene una soluzione codificata nel suo DNA (stringa binaria): valutando la bontà della soluzione fornita da ciascun *individuo* mediante l'applicazione di una **funzione costo**[2], si selezionano gli individui che si dovranno riprodurre.

La riproduzione delle stringhe avviene, in analogia con gli esseri viventi a riproduzione sessuata, tramite **accoppiamento**, cioè tramite la fusione delle due stringhe appartenenti ai genitori.

In pratica ciò si realizza stabilendo un punto in cui spezzare in due ciascuna delle stringhe, per accoppiare poi ciascuna metà con la metà complementare dell'altra stringa.

I discendenti così prodotti non vanno a rimpiazzare i genitori, che, ricordiamolo, sono le stringhe che hanno fornito i migliori valori per la funzione costo; essi rimpiazzano invece le stringhe della precedente generazione che hanno dato valori meno buoni della funzione costo.

In questo modo la popolazione globale non cambia ad ogni generazione.

Quindi, al fine di evitare la produzione di popolazioni troppo uniformi (che esplorano una porzione ristretta dello spazio delle soluzioni), si produce su una piccola percentuale di individui una **mutazione** (per esempio negando il valore di un bit della stringa).

[2]Con il termine *funzione costo* si indica una funzione che, data una soluzione ammissibile del problema, restituisce un valore che ne rappresenta la "bontà ".

Nelle prossime sezioni vedremo sia come funzionano gli AG che un importante risultato noto come teorema degli schemi.

1.1 Funzionamento degli AG

Gli algoritmi genetici simulano lo sviluppo di una popolazione di individui usando il metodo della selezione naturale e lo scambio di materiale genetico. Gli AG non sono legati alle particolari caratteristiche del problema a cui sono applicati. Essi forniscono ottimi locali e attualmente non è disponibile nessuna informazione teorica sull'errore della soluzione trovata rispetto all'ottimo globale.

Gli AG si differenziano da altri algoritmi di approssimazione su vari punti. Alcuni di questi sono:

- non trattano direttamente con gli elementi dello spazio ammissibile, ma con una loro codifica (in alcuni casi la funzione di codifica può essere l'identità);
- trattano una popolazione di elementi, non singoli elementi;
- usano regole di transizione probabilistiche per la generazione di ogni popolazione.

Ogni parametro del problema da ottimizzare è codificato in una stringa di bit, che sono visti come un **gene**. L'insieme dei parametri è codificato in una stringa di bit, che sono visti come un **cromosoma**. L'algoritmo non ha nessuna conoscenza sul significato della stringa di bit.

Gli algoritmi genetici non lavorano a partire da un singolo punto nello spazio di ricerca, ma da una popolazione di individui. Negli algoritmi genetici ogni soluzione è vista come un individuo della popolazione, l'insieme di soluzioni all'istante t viene detto **popolazione**. Ad ogni individuo è assegnato un valore di **fitness**[3]. Quest'ultimo non deve riguardare solo un attributo di un particolare gene, piuttosto la combinazione totale dei geni.

Gli algoritmi genetici sono stati introdotti da Holland in [23]. L'algoritmo presentato da Holland è riportato in Figura 1.1. Con P_C si indica la probabilità di applicare l'operatore di crossover, con P_M la probabilità di mutare i geni dell'individuo trattato e con P_I la probabilità di invertire la sequenza dell'intero codice genetico di un individuo.

L'algoritmo di Holland funziona come segue. Sia $\beta(0)$ la popolazione iniziale generata casualmente e $\beta(t)$ la popolazione di m individui all'istante t, l'algoritmo iterativamente genera una nuova popolazione $\beta(t+1)$ a partire da $\beta(t)$ applicando alcuni operatori genetici. Questi operatori modificano alcuni individui nella popolazione $\beta(t)$ ottenendo così una popolazione $\beta(t+1)$ che contiene un maggior numero di individui che hanno un valore di $fitness$ migliore.

La condizione di terminazione può essere fissata da:

- un numero massimo di iterazioni, raggiunto il quale l'algoritmo si ferma;
- un controllo sul miglioramento del valore di $fitness$ fornita dalla soluzione corrente; quando si notano miglioramenti poco apprezzabili delle soluzioni, l'algoritmo è interrotto.

Nei paragrafi successivi è descritto ogni singolo passo dell'algoritmo.

[3]La $fitness$ indica quanto è buona la soluzione fornita dall'individuo. Tipicamente, coincide con il valore che si ottiene calcolando la funzione costo sulla soluzione.

```
Program Algoritmo_Genetico;
begin
t=0;
β (t) = INIZIALIZZA_POPOLAZIONE() ;
repeat
    for i = 1 to m do f(A_i) = COMPUTA_FITNESS(A_i);
    fitness_medio = CALCOLA_FITNESS_MEDIO(β (t));
    for k = 1 to m do
    begin
        A_k = SELEZIONA(β (t));
        /* Ogni individuo viene selezionato confrontando il suo valore di
        fitness con il fitness_medio */
        if (P_C > random(0,1)) then
        begin
            A_i = SELEZIONA(β (t));
            A_figlio = CROSSOVER( A_i,A_k);
            if (P_M > random (0,1)) then MUTAZIONE
            (A_figlio);
            if (P_I > random(0,1)) then INVERSIONE
            (A_figlio);
            β (t + 1)=AGGIORNA_POPOLAZIONE (A_figlio);
        end
    end;

t=t+1;
until (condizione_di_terminazione);
end
```

Figura 1.1: Algoritmo genetico di Holland.

1.2 Operatori degli AG

1.2.1 Selezione

L'operatore di selezione ha la funzione di scegliere gli individui ai quali devono essere applicati gli operatori genetici. La selezione degli individui può essere fatta casualmente, oppure per mezzo del valore di $fitness$. In quest'ultimo caso gli individui con valore di $fitness$ migliore hanno una probabilità più alta di essere selezionati per generare i figli.

Questa operazione di selezione degli individui con valore di $fitness$ più alto, può essere vista come una metafora della sopravvivenza naturale di Darwin. Infatti in natura il valore di $fitness$ di un individuo è determinato dall'abilità di superare tutti gli ostacoli dell'ambiente.

La fase di selezione è applicata sulla *popolazione corrente*. Le stringhe selezionate sono messe in una *popolazione intermedia*. Gli operatori genetici sono applicati alla popolazione intermedia, per creare la *popolazione successiva*. Ottenuta quest'ultima si dice allora che nell'esecuzione dell'algoritmo genetico è trascorsa una generazione.

1.2.2 Rimpiazzamento

Ci sono due modi principali per rimpiazzare un individuo nella popolazione:

- *rimpiazzamento generazionale*;
- *rimpiazzamento a stato continuo.*

Nel *rimpiazzamento generazionale* la popolazione corrente è rimpiazzata dalla successiva; in questo caso le dimensioni della popolazione corrente e della successiva devono essere uguali.

Nel *rimpiazzamento a stato continuo*, i nuovi individui generati rimpiazzano gli individui peggiori della popolazione corrente quando i figli hanno ottenuto un valore di *fitness* migliore. La popolazione successiva è identica alla popolazione corrente eccetto per l'ultimo figlio generato. In letteratura questo modello è conosciuto come *modello statico di popolazione*.

Un'importante differenza tra i due metodi è che nel *rimpiazzamento a stato continuo* i nuovi figli sono immediatamente disponibili per la riproduzione, perciò gli AG hanno l'opportunità di esplorare subito gli individui più promettenti appena essi vengono creati. Questi due modelli di rimpiazzamento vengono impiegati coerentemente col modello di generazione adottato: il *rimpiazzamento generazionale* col *modello di generazione discreto* e il *rimpiazzamento a stato continuo* col *modello di generazione continuo*. I modelli di generazione sono descritti nel Paragrafo 1.3.

1.2.3 Crossover

L'operatore di *crossover* prende parti dei due genitori e crea un nuovo individuo (figlio) combinando le parti prese. In questo modo i figli contengono informazioni genetiche di entrambi i padri.

I punti di *crossover* (di solito due) sono scelti casualmente. I nuovi individui consisteranno di parti alternate delle stringhe dei genitori.

Nonostante sia casuale, questo scambio di informazione genetica rende gli algoritmi genetici molto versatili; infatti una parte buona di un genitore può rimpiazzare una parte meno buona dell'altro genitore, creando un figlio migliore dei genitori. Un esempio di applicazione del *crossover* è mostrato in Figura 1.2.

1.2.4 Inversione

L'operatore di inversione prende due punti casuali nel cromosoma e inverte la sequenza di geni tra questi due punti. L'importanza del cambiamento

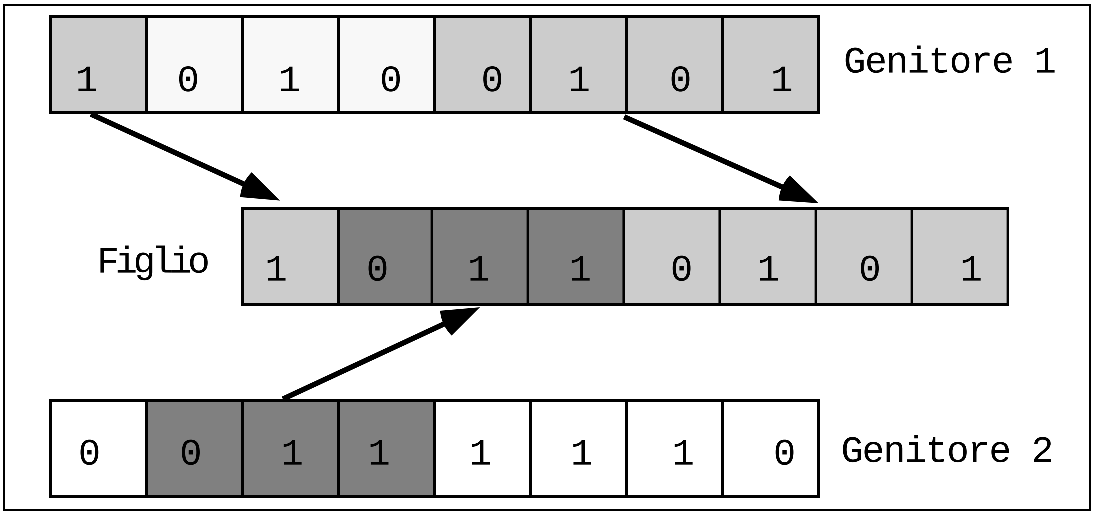


Figura 1.2: Esempio di funzionamento dell'operatore di *crossover*.

così effettuato dipende dalla particolare codifica scelta per rappresentare la soluzione. In Figura 1.3 è mostrato un esempio di uso dell'operatore di inversione.

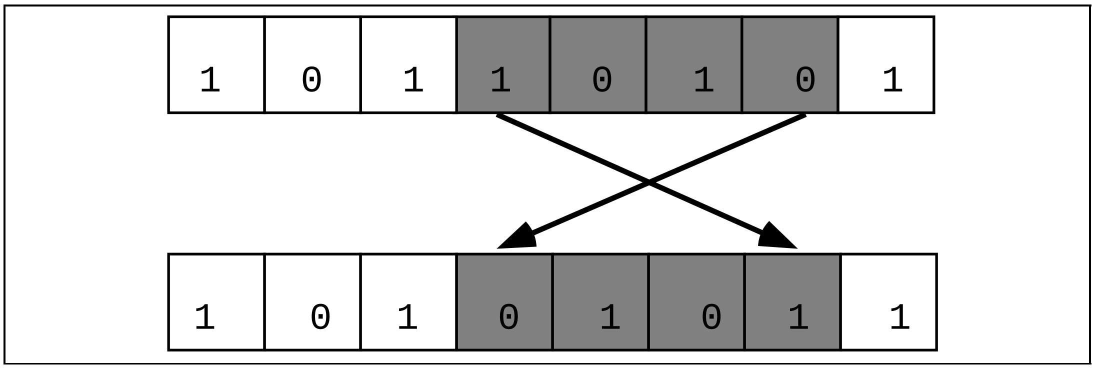


Figura 1.3: Esempio di applicazione dell'operatore genetico di inversione.

1.2.5 Mutazione

L'operatore di mutazione è molto semplice. Esso cambia casualmente i valori di un gene in un individuo appena creato; nel caso della codifica binaria della soluzione, si nega il valore di un bit. È impiegato principalmente per introdurre delle varianti nelle soluzioni della popolazione corrente, che consentano di sfuggire dagli ottimi locali quando la maggioranza delle soluzioni converge verso questi ultimi.

1.3 Modello di generazione

Il modello di generazione descrive il modo secondo il quale la popolazione si evolve.

Come risulta da [20], esistono due modelli di generazione:

- **modello di generazione discreto**;
- **modello di generazione continuo**.

Nel *modello di generazione discreto* (vedi Figura 1.4) si tiene ogni generazione ben distinta dall'altra. Infatti la generazione successiva $\beta(t+1)$, formata dai figli, è generata dalla popolazione dei genitori $\beta(t)$.

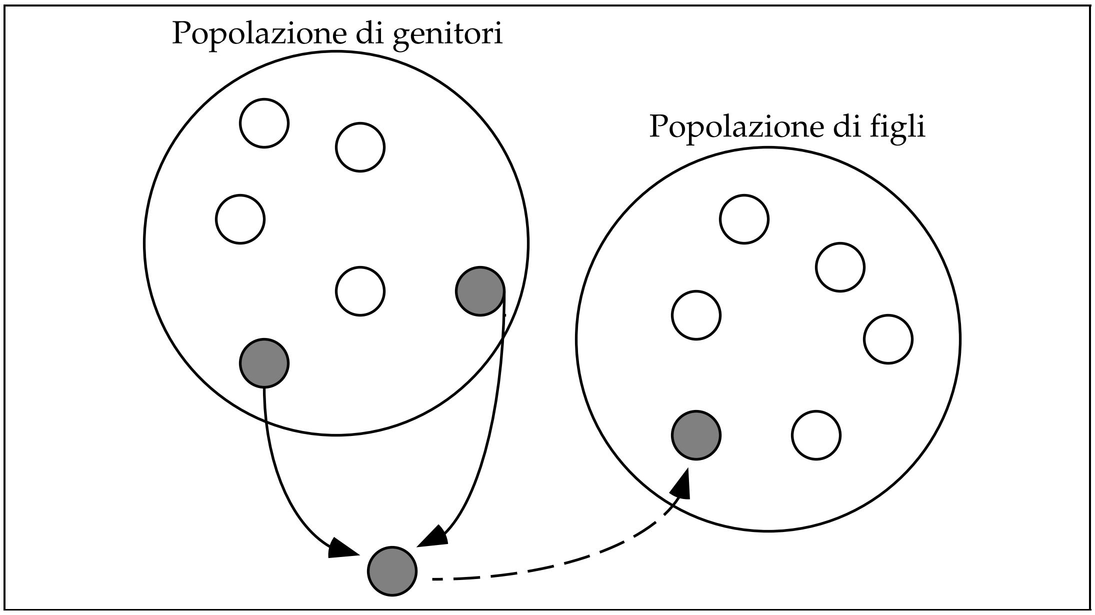


Figura 1.4: Modello di generazione discreto.

In questo modo riusciamo a distinguere le popolazioni di genitori da quella dei figli. Se tutti i genitori hanno prodotto i loro figli, questi ultimi diventeranno i nuovi genitori della successiva generazione e così via.

Un caso generale è stato introdotto in [26] in cui l'autore ha inserito un parametro $0 < G \leq 1$ che controlla la percentuale di popolazione da trattare durante ogni generazione. Il numero di individui selezionati è pari alla dimensione della popolazione moltiplicato per il parametro G. Questo modello è stato chiamato *a sovrapposizione di popolazione* perchè solo alcuni individui della popolazione corrente sono selezionati per l'applicazione degli operatori genetici. I rimanenti individui sopravvivono intatti nella prossima generazione. A G=1 corrisponde il modello di generazione discreto.

Nel *modello di generazione continuo* (vedi Figura 1.5) la decisione sulla sopravvivenza di un figlio viene fatta immediatamente dopo la sua creazione, anzichè dopo la generazione di tutti gli altri figli. I figli diventano immediatamente disponibili come genitori. Quindi non è possibile distinguere nella generazione corrente tra genitori e figli.

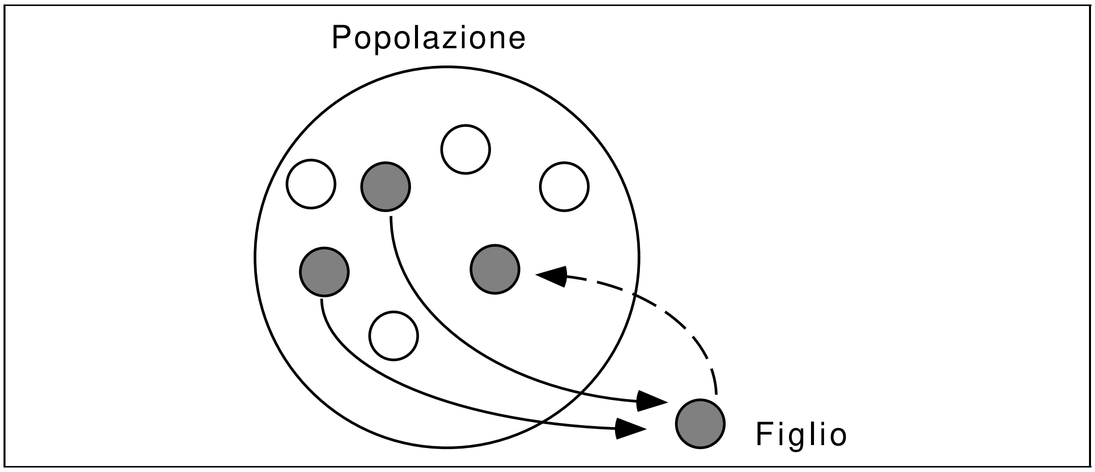


Figura 1.5: Modello di generazione continuo.

1.4 Aspetti teorici degli AG

Gli aspetti teorici trattati nei seguenti paragrafi si basano sulla definizione originaria di AG dovuta a Holland [23]. Se si immagina di disporre nello spazio tutte le stringhe binarie di lunghezza n possibili, ponendo più vicine quelle che differiscono per il minor numero di bit, si possono individuare alcune *regioni* che forniscono risulati migliori di altre per la funzione costo.

Per esempio, su una stringa di 4 bit, si potrebbe trovare che la regione identificata da 1*0* (dove * sta per 0 o 1 indifferentemente) fornisce in genere buoni valori per la funzione costo.

Gli algoritmi genetici riescono a campionare stringhe appartenenti a varie regioni, e attraverso la selezione degli individui migliori, si concentrano attorno alle regioni più promettenti.

Il numero di stringhe appartenenti a regioni con un buon valore della funzione costo aumenta di generazione in generazione: la fusione di stringhe (che avviene tramite l'operatore di *crossover*) potrebbe anche trasferire un discendente da una regione all'altra. Ciò si riflette in un rallentamento dell'evoluzione verso una buona soluzione.

La probabilità che i discendenti di due stringhe si allontanino dalla regione dei genitori dipende dalla distanza tra gli 0 e gli 1 che definiscono quella regione. I discendenti di una stringa della regione 1**0 rischiano di allontanarsi da tale regione indipendentemente da dove si situi il punto di separazione, mentre i discendenti della regione 10** hanno migliori possibilità di rimanere accanto ai loro genitori. I blocchi di 0 e 1 adiacenti in una regione si chiamano **blocchi costitutivi compatti**.

L'importanza dei *blocchi costitutivi compatti* deriva dal fatto che è molto probabile che essi rimangano intatti durante il processo di fusione di stringhe e quindi che si ritrovino in generazioni future proporzionalmente alla bontà della funzione costo posseduta dalle stringhe che li trasportano.

Il meccanismo fin qui illustrato riesce a campionare le regioni proporzionalmente alla bontà della loro funzione costo, solo in presenza di regioni definite da blocchi compatti.

Negli organismi viventi esiste un processo chiamato inversione, il quale ridispone occasionalmente i geni in maniera tale che quelli che si trovano più distanti nei genitori si ritrovino più vicini nella prole. Ciò corrisponde a ridefinire un blocco costitutivo in modo che sia meno soggetto alla separazione durante la fusione di stringhe. Ciò consente di perdere un minor numero di discendenti, e quindi di concentrarsi adeguatamente sulle zone più promettenti dello spazio delle stringhe.

Quanto sopra descritto è formalizzato in termini matematici tramite il **teorema degli schemi**.

1.4.1 Teorema degli schemi

Come risulta da [14], il teorema è basato su tre definizioni:

- **schema**;
- **lunghezza** di uno schema;
- **ordine** di uno schema.

Lo *schema* è una successione di simboli $1, 0, *$. Uno schema rappresenta tutte le stringhe che hanno i bit uguali in tutte le posizioni diverse da quelle occupate dal simbolo $*$. La nozione di schema permette di studiare le somiglianze tra i cromosomi.

Per esempio lo schema:

$$F = (*, 1, 1, 1, 0, 0, 0, 1, 0, 0)$$

rappresenta le stringhe:

$$(0, 1, 1, 1, 0, 0, 0, 1, 0, 0)$$

$$(1, 1, 1, 1, 0, 0, 0, 1, 0, 0).$$

Ogni *schema* con r simboli *, rappresenta 2^r stringhe di bit, diverse tra loro.

Una stringa può appartenere a diversi schemi: per esempio la stringa

$$(1, 1, 1, 1, 0, 0, 0, 1, 0, 0)$$

appartiene ad entrambi gli schemi:

$$(*, *, *, 1, 0, 0, 0, 1, *, *)$$

e

$$(1, 1, 1, *, *, *, *, *, 0, 0).$$

Dal momento che ogni *schema* esplora una particolare *regione* dello spazio delle soluzioni, un AG che manipola una popolazione di poche centinaia di stringhe dispone di campioni di un numero molto più grande di *regioni*. In breve gli AG elaborando esplicitamente una stringa, elaborano implicitamente più schemi, introducendo in tal modo un **parallelismo implicito**.

L'*ordine* di uno schema S, indicato con $o(S)$, è il numero di 0 o 1 presenti nello schema. Per esempio lo schema F ha $o(F) = 9$.

La *lunghezza* di uno schema S, indicata con $\Delta(S)$, è la distanza tra la prima e l'ultima posizione che abbiano valore 0 oppure 1. Per esempio lo schema

$$(*, *, *, 1, 0, 1, *, *, 1, 1, 1, *, 0, *)$$

ha lunghezza 10. Tale nozione sarà utile nel calcolare la probabilità di sopravvivenza di uno schema all'operatore di *crossover* (che tende a spezzare gli schemi). L'introduzione degli schemi consente di analizzare quantitativamente l'effetto della riproduzione e degli operatori genetici su una popolazione di cromosomi.

Supponiamo che ad un certo istante t ci siano m esemplari di uno schema S (ovvero di soluzioni rappresentate dallo schema S) che indichiamo con $m(S,t)$. Se un individuo i ha un valore di $fitness$ f_i, possiede una probabilità

$$p_i = \frac{f_i}{FT} \tag{1.1}$$

di essere selezionato, dove FT è il valore $fitness$ totale della popolazione, cioè la sommatoria di tutti i $fitness$ ($FT = \sum_{i=1}^{n} f_i$).

Se n è il numero di individui della popolazione, il numero atteso di individui rappresentati dallo schema S nella successiva popolazione applicando l'operatore di selezione sarà dato da:

$$m(S,t+1) = m(S,t) \cdot n \cdot \frac{f(S)}{FT} \tag{1.2}$$

dove $f(S)$ è il valore di $fitness$ medio dei cromosomi corrispondenti allo schema S all'istante t.

Se indichiamo con $f = \frac{FT}{n}$ l'adattamento medio della popolazione, possiamo scrivere:

$$m(S,t+1) = \frac{m(S,t) \cdot f(S)}{f} \tag{1.3}$$

Quindi, la velocità di sviluppo di uno schema è data dal rapporto tra l'adattamento medio dello schema e l'adattamento medio della popolazione complessiva.

Gli schemi con adattamento medio superiore alla media della popolazione, ricevono un numero crescente di cromosomi al crescere delle generazioni;

viceversa, uno schema che si trova sotto la media di adattamento vedrà diminuire il proprio numero di cromosomi. Comunque la selezione da sola non consente di introdurre nuovi cromosomi: questo compito spetta agli operatori di *crossover* e di mutazione.

Discutiamo ora l'effetto di questi due operatori sul numero di individui atteso per uno schema della popolazione.

Applicando l'operatore di *crossover* ad un punto scelto casualmente, la probabilità $p_d(S)$ che uno schema S venga distrutto coincide con la probabilità che il punto di *crossover* cada all'interno della sua lunghezza, tale probabilità è data da:

$$p_d(S) = \frac{\Delta(S)}{l-1} \tag{1.4}$$

dove l è la lunghezza della stringa, mentre la probabilità di sopravvivenza è data da:

$$p_s(S) = 1 - p_d(S) \tag{1.5}$$

Tenendo conto che il *crossover* viene applicato con probabilità p_c, otteniamo la seguente relazione:

$$p'_s(S) = 1 - p_c \cdot \frac{\Delta(S)}{l-1} \tag{1.6}$$

Applicando l'operatore di mutazione uno schema viene distrutto quando il punto di mutazione coincide con la posizione di un bit di valore 1 o 0 nello schema. Sia p_m la probabilità di alterazione di un singolo bit, la probabilità che un singolo bit non venga alterato è $1 - p_m$.

Una mutazione singola è indipendente dalle altre, perciò la probabilità che uno schema S non venga distrutto si ottiene elevando la probabilità che un bit non venga mutato ad un esponente uguale all'ordine dello schema. Si ha così:

$$p_{sm}(S) = (1 - p_m)^{o(S)} \tag{1.7}$$

Per piccoli tassi di mutazione la formula precedente può essere approssimata con $1 - p_m \cdot o(S)$.

Combinando gli effetti di riproduzione, *crossover* e mutazione si ottiene la seguente formula:

$$m(S, t+1) = m(S, t) \cdot \frac{f(S)}{f} \cdot [1 - p_c \cdot \frac{\Delta(S)}{l-1} - p_m \cdot o(S)] \qquad (1.8)$$

La formula precedente esprime il teorema **degli schemi**.

Teorema degli schemi . *Schemi di piccola lunghezza, di basso ordine, con adattamento sopra la media vengono verificati*[4] *con una frequenza esponenzialmente crescente nelle generazioni successive di un AG.*

Una ipotesi diffusa tra gli studiosi di AG è la seguente:

Ipotesi dei blocchi costitutivi . *Un AG persegue soluzioni vicine all'ottimo mediante la giustapposizione di schemi di piccola lunghezza, di basso ordine, di adattamento sopra la media, chiamati* **blocchi costitutivi**.

1.5 Conclusioni

In questo capitolo abbiamo illustrato i principali aspetti degli algoritmi genetici. Si possono distinguere alcuni aspetti positivi e altri negativi.

Gli aspetti positivi sono:

- parallelismo implicito, per cui la quantità di informazione trattata è maggiore del numero di individui effettivamente elaborati;

[4] Uno schema è verificato quando nuove soluzioni vengono aggiunte a quelle rappresentate dallo schema stesso.

- facilità di adattamento dell'algoritmo alla risoluzione di una vasta classe di problemi;

Gli aspetti negativi sono:

- risultati teorici ancora limitati.

Capitolo 2

Algoritmi genetici paralleli

Nel capitolo precedente abbiamo illustrato il funzionamento degli algoritmi genetici sequenziali.

La maggiore disponibilità di elaboratori ad alto parallelismo ha reso possibile la definizione e l'utilizzo di AG paralleli con lo scopo di diminuire il tempo di esecuzione ottenendo la stessa qualità delle soluzioni.

Vari modelli di parallelizzazione sono elencati in [8] e [6], e sono suddivisi in modelli:

- **a grana fine**: si opera su una singola popolazione in cui ogni individuo è posizionato in un elemento della topologia della struttura di interconnessione usata. Gli operatori genetici vengono applicati solamente tra individui vicini;
- **a grana grossa**: varie sottopopolazioni si evolvono in parallelo, scambiandosi periodicamente gli individui che forniscono soluzioni migliori;
- **centralizzata**: una popolazione **panmitica** [1] è elaborata da vari processi, alcuni dei quali, i processi *slave*, realizzano gli operatori genetici

[1] Si dice **panmitica** una popolazione priva di struttura, nella quale ogni individuo si può accoppiare con qualsiasi altro individuo.

classici, mentre un solo processo, chiamato *master*, seleziona ed invia agli *slave* gli individui che forniscono soluzioni migliori.

2.1 Modelli di parallelizzazione

2.1.1 Algoritmi genetici paralleli a grana fine

Questo modello sfrutta i concetti di **spazialità** e di **vicinanza**. Il primo termine prevede che la popolazione di individui sia distribuita secondo una topologia di interconnessione logica, mentre il secondo specifica che l'accoppiamento e la selezione avvengono solamente tra individui vicini nella struttura topologica utilizzata. Un esempio di vicinanza è riportato in Figura 2.1.

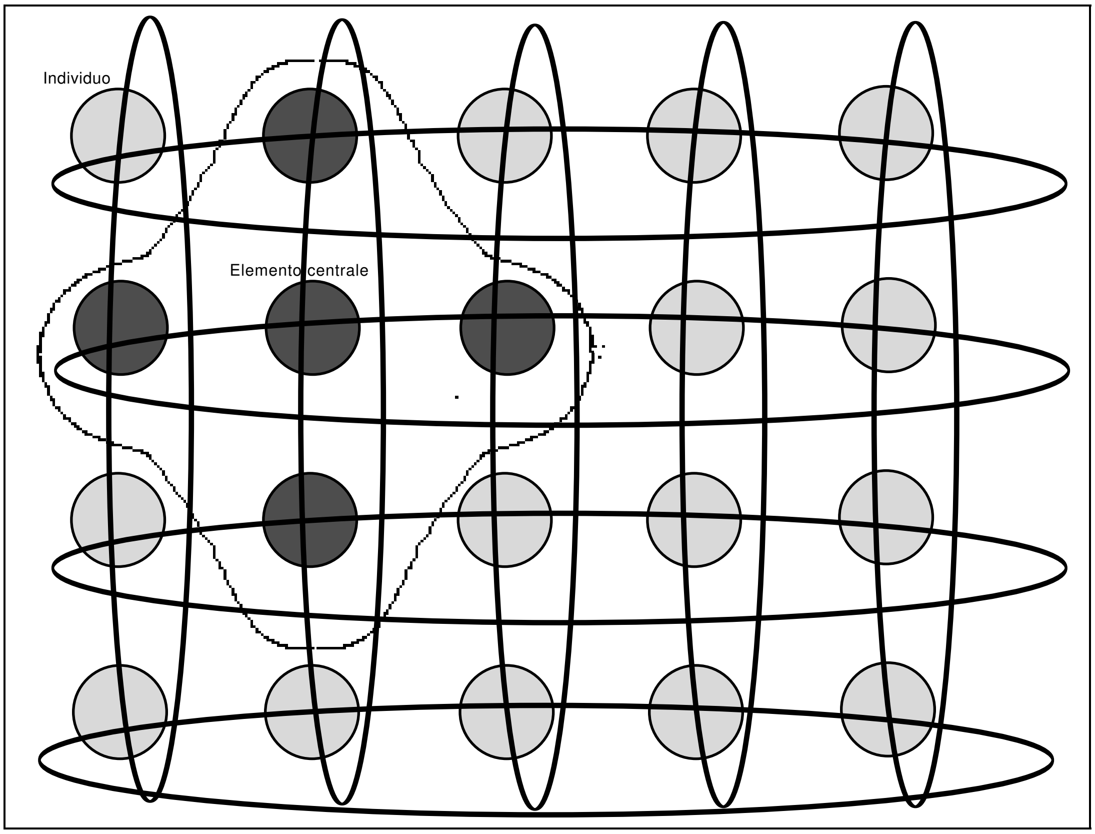


Figura 2.1: Struttura logica toroidale di connessione tra individui.

L'insieme dei vicini di un individuo è detto il suo **intorno** e determina tutti i suoi potenziali **partner** per la riproduzione. È importante notare che, secondo il modello di vicinanza tra individui utilizzato, uno stesso individuo può appartenere contemporaneamente a più intorni dei suoi vicini, e quindi può accoppiarsi indifferentemente con ciascuno di essi (Figura 2.2).

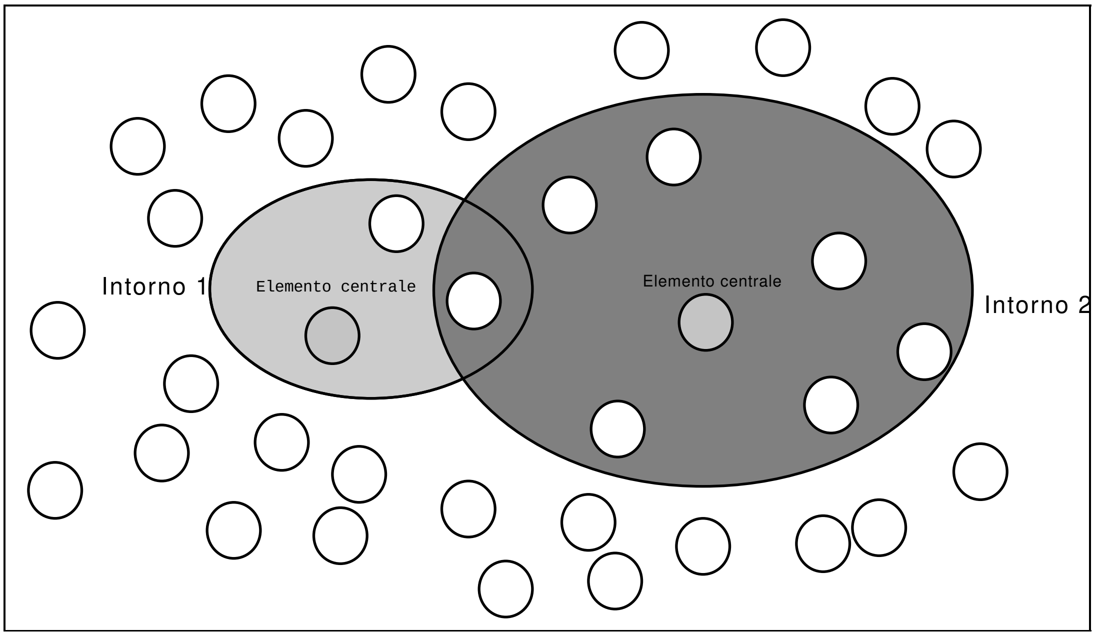


Figura 2.2: Esempio di sovrapposizione di intorni.

Le comunicazioni tra i nodi devono avvenire rispettando il modello logico di vicinanza, cioè devono intercorrere solamente tra nodi logicamente vicini.

Possiamo sintetizzare le proprietà principali del modello a grana fine nei seguenti punti:

- nessun controllo centralizzato;
- iterazioni solo tra gli individui *vicini*;
- indipendenza di ogni singolo individuo.

```
Program Grana_Fine;
begin
popolazione_iniziale=GENERA(numero_individui);
ASSEGNA_POSIZIONE_NELLA_GRIGLIA(popolazione_iniziale);
while condizione_di_terminazione do

        for cella = 1 to numero_celle do
        begin

                contenuto_cella=SELEZIONA(popolazione);
                /* Seleziona l'individuo che deve occupare la posizione cella della griglia, tra
                quelli assegnati a cella e al suo intorno */
                individuo_vicino=SELCASUALE(popolazione);
                CROSSOVER(individuo_vicino,contenuto_cella);
                /* Seleziona casualmente un individuo tra i vicini di cella ed effettua un
                crossover con la soluzione associata a cella secondo la probabilità di ripro-
                duzione. Assegna uno dei figli a contenuto_cella */
                MUTAZIONE(contenuto_cella);
                /* Muta l'individuo assegnato a contenuto_cella secondo la probabilità di mu-
                tazione */
                COMPUTA_FITNESS(contenuto_cella);
                /* Calcola il valore della fitness del nuovo individuo assegnato a cella */

        end;

end
```

Figura 2.3: Algoritmo a grana fine espresso in pseudocodice.

Un classico algoritmo a grana fine è quello descritto in [14], il cui pseudocodice è riportato in Figura 2.3.

2.1.2 Implementazioni del modello a grana fine

In [19] è descritto il sistema ASPARAGOS. L'autrice usa una struttura di interconnessione che assomiglia ad una scala con gli estremi che combaciano, come riportato in Figura 2.4.

ASPARAGOS è stato usato con successo per risolvere vari problemi di ottimizzazione combinatoriale. Dallo studio del caso trattato emergono alcune indicazioni sull'influenza dei parametri di mutazione, *crossover* e numerosità della popolazione nel funzionamento dell'algoritmo genetico: un alto tasso di mutazione rallenta la convergenza, il punto di *crossover*, invece, non incide molto (quando è mantenuto nell'intervallo tra $N/2$ e $N/3$, con N la lunghezza della codifica della soluzione), popolazioni più numerose hanno una probabi-

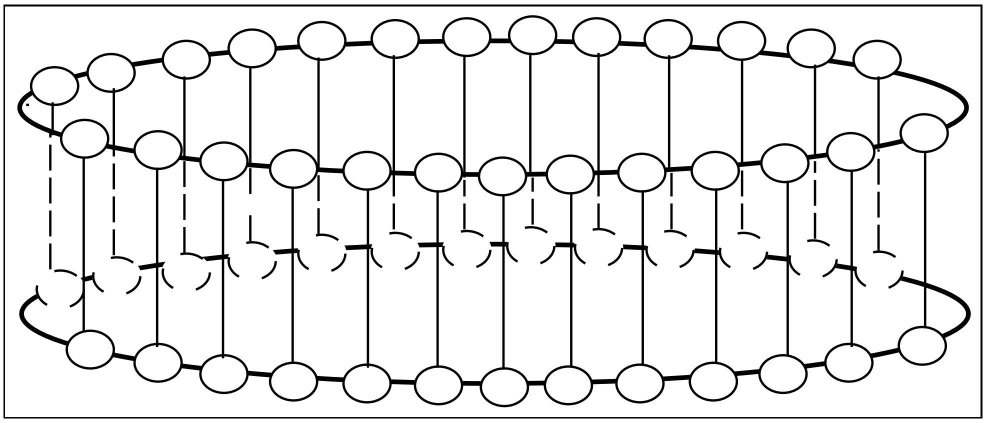


Figura 2.4: Struttura a scala del sistema ASPARAGOS.

lità più alta di ottenere soluzioni migliori. Inoltre un'alta connettività della topologia di interconnessione sembra portare verso una veloce predominanza delle soluzioni migliori sulle altre, comportando quindi una convergenza più veloce verso ottimi locali.

In [41] è stata proposta una variazione dell'algoritmo ASPARAGOS. Purtroppo, a causa dell'utilizzo di una procedura di ricerca locale (vedi capitolo 3) che migliora gli individui della popolazione, è difficile stabilire se i miglioramenti ottenuti dipendono dalla bontà dell'algoritmo genetico, o piuttosto dalla procedura di ricerca locale utilizzata. L'algoritmo proposto è riportato in Figura 2.5.

Un'altra implementazione è stata proposta in [34] da Manderick e Spiessens. In questo algoritmo la popolazione è distribuita su una topologia logica di interconnessione bidimensionale. Nello studio si rileva che le prestazioni dell'algoritmo degradano al crescere del numero di vicini di ogni individuo. Se il concetto di intorno di un individuo comprende abbastanza vicini, l'algoritmo si comporta come una popolazione panmitica (priva di struttura). Problemi semplici sono risolti meno efficientemente, dal momento che la grossa potenzialità esplorativa dell'algoritmo si concretizza in un carico compu-

```
Program ASPARAGOS_CON_RICERCA_LOCALE;
begin
popolazione=GENERA(numero_individui);
while condizione_di_terminazione do
begin

    RICERCA_LOCALE(popolazione);
    for i = 1 to numero_individui do
    begin

        individuo=SELEZIONA(popolazione);
        individuo_vicino=SELEZIONA_VICINO(popolazione);
        figlio=CROSSOVER(individuo_vicino,individuo);

    end;
    RIMPIAZZAMENTO(popolazione,figli);
    COMPUTA_FITNESS(contenuto_cella);

end;
end
```

Figura 2.5: Algoritmo ASPARAGOS con ottimizzatore locale.

tazionale ridondante rispetto, per esempio, ad una semplice ricerca locale. Il lavoro conclude che, nel caso di implementazione su elaboratori paralleli, al crescere della popolazione, cresce anche la bontà della soluzione trovata.

Schwehm [48] confronta i risultati di implementazioni di algoritmi genetici a grana fine effettuati su diverse topologie di interconnessione. I risultati mostrano che la topologia toroidale permette di ottenere i risultati migliori.

2.1.3 Algoritmi genetici paralleli a grana grossa

Nel modello a grana grossa la popolazione è divisa in un certo numero di sottoinsiemi chiamati **isole**, che determinano una partizione della popolazione complessiva in sottopopolazioni.

Il processo evolutivo avviene esclusivamente all'interno di tali isole. In questo modo, rispetto ad una popolazione panmitica senza struttura, si ha la possibilità di mantenere una maggiore diversità genetica all'interno della popolazione complessiva. Ciò si riflette in una migliore esplorazione dello spazio delle soluzioni: isole diverse esplorano zone distanti in parallelo. Per evitare la convergenza delle popolazioni delle isole verso ottimi locali, si effettua periodicamente una migrazione di alcuni individui di un'isola verso un'altra. In tal modo si rimescolano i geni delle popolazioni, e si evita una convergenza su un ottimo locale.

I due principali modelli di implementazione per l'algoritmo a grana grossa sono:

- il modello **a isole**;
- il modello **stepping stone**.

Mentre nel modello ad *isole* (vedi Figura 2.6) la migrazione avviene verso qualsiasi isola, nel modello *stepping stone* (vedi Figura 2.7) si definisce una vicinanza tra isole, cosicchè la migrazione possa avvenire esclusivamente tra isole vicine.

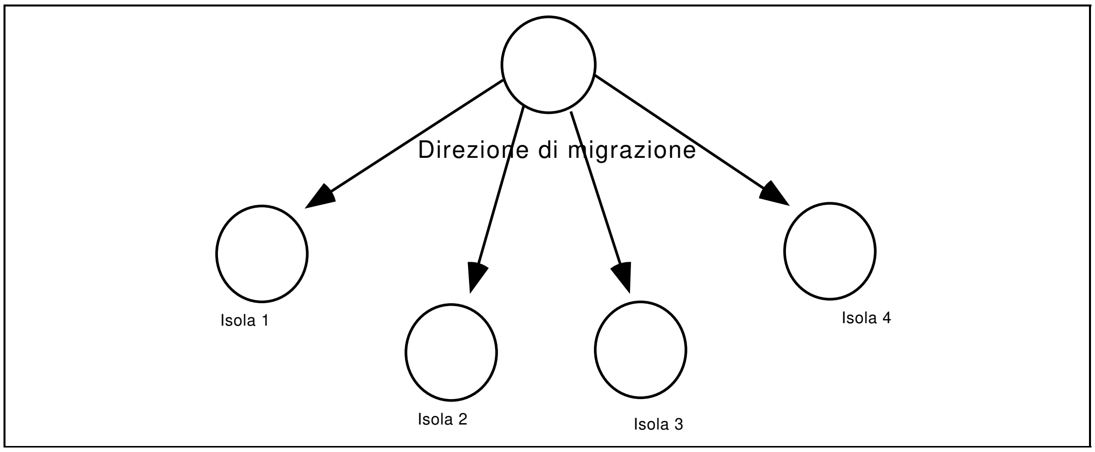


Figura 2.6: Esempio di implementazione ad *isole*.

I risultati degli studi fin qui condotti mostrano come il numero di individui fatti migrare e l'intervallo di migrazione (intervallo di tempo tra due migrazioni successive) siano entrambi fattori critici: un alto numero di individui che migrano riducono il comportamento del modello a isole a quello di

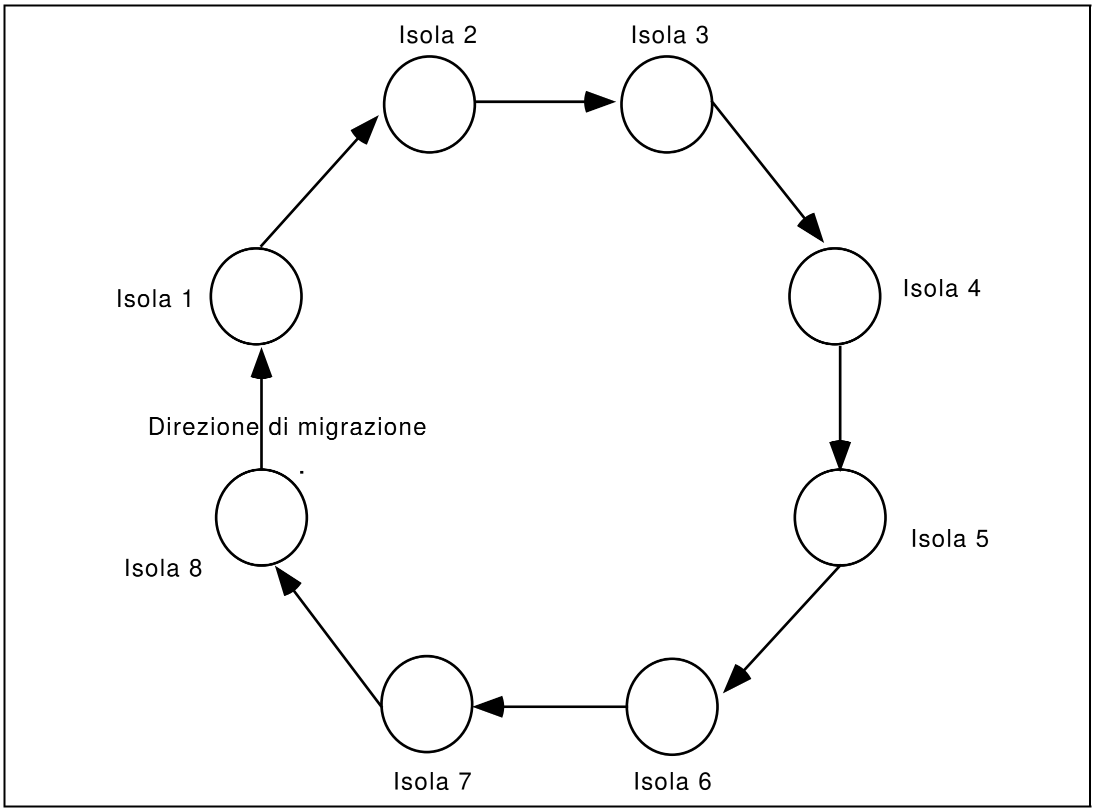


Figura 2.7: Esempio di implementazione a *stepping stone*.

una popolazione panmitica, e quindi si perdono i vantaggi rispetto a quest'ultima. Un basso numero di migranti tendono a non rimescolare il patrimonio genetico, e quindi non permettono di oltrepassare gli ottimi locali trovati nelle singole isole.

2.1.4 Implementazioni del modello a grana grossa

Uno dei primi studi sul modello a grana grossa è quello di Grosso [22]. La popolazione, suddivisa in cinque sottopopolazioni, effettua migrazioni di un numero fissato e sempre uguale di individui. Grosso ha verificato che il tasso di miglioramento delle soluzioni era più rapido rispetto ad una popolazione panmitica. Quando le sottopopolazioni sono lasciate completamente isolate la soluzione risulta peggiore di quella ottenuta con una popolazione panmitica. Con un numero intermedio di individui migrati, si verifica un comportamento simile al panmitico, mentre, con un minor numero di individui migrati, le sottopopolazioni hanno la possibilità di evolversi indipendentemente e di esplorare regioni diverse dello spazio delle soluzioni.

Questi esperimenti mostrano l'esistenza di un numero critico di individui da far migrare, al di sotto del quale la *performance* dell'algoritmo peggiora; al di sopra di tale numero critico di individui l'evoluzione avviene in maniera molto simile al modello panmitico.

Tanese in [49] ha proposto un algoritmo genetico parallelo che usa una topologia a ipercubo a quattro dimensioni per comunicare gli individui da una sottopopolazione all'altra. Nella implementazione di Tanese le migrazioni avvengono, dopo un numero costante di generazioni, tra sottopopolazioni vicine. Gli individui da trasferire sono scelti probabilisticamente tra gli individui migliori e rimpiazzano gli individui peggiori della sottopopolazione destinazione.

I principali parametri sui quali Tanese ha effettuato i suoi esperimenti sono: popolazione totale di 400 individui, tasso di *crossover* pari a 0.6, tasso di mutazione di 0.5 e uno scambio del 10% della sottopopolazione ogni 5 generazioni.

I risultati sperimentali indicano che l'algoritmo parallelo, in termini di numero di generazioni necessarie a trovare l'ottimo è paragonabile a quello seriale, e che lo *speedup* ottenuto ha un andamento molto vicino a quello lineare. Un esperimento successivo ha permesso di verificare che l'algoritmo riesce a trovare l'ottimo in un numero di generazioni compreso tra 150 e 200 anche in presenza di tassi di mutazione, *crossover* e migrazione diversi su ogni processore.

```
Program AG_SEQUENZIALE;
begin
popolazione=GENERA (pop_size);
for gen = 1 to num_gens do
begin

        popolazione_fitness=CALCOLA_FITNESS(popolazione);
        for i = 1 to (pop_size/2) do
        begin

                primo_genitore=SELEZIONA(popolazione);
                secondo_genitore=SELEZIONA(popolazione);
                primo_figlio=CROSSOVER(primo_genitore,secondo_genitore);
                secondo_figlio=CROSSOVER(secondo_genitore,primo_genitore);
                /* Il crossover produce dai medesimi genitori due figli differenti */
                MUTAZIONE(primo_figlio);
                MUTAZIONE(secondo_figlio);

        end;
        INSERISCI_MIGLIOR_INDIVIDUO(popolazione);
        /* Se l'individuo con migliore valore di fitness nella precedente popolazione non è presente nella
        nuova popolazione allora lo si inserisce togliendo un individuo da quest'ultima */

end;
end
```

Figura 2.8: Algoritmo genetico sequenziale di Tanese.

Tanese conclude che l'algoritmo parallelo opera altrettanto bene di quello seriale, con il vantaggio di uno *speedup* vicino alla linearità.

L'algoritmo sequenziale di Tanese è riportato in Figura 2.8; l'algoritmo parallelo è riportato in Figura 2.9.

Tanese ha anche studiato l'effetto dell'uso di differenti tassi di migrazione sulla *performance* dell'algoritmo (numero di individui trasferiti e frequenza di migrazione). Nelle conclusioni, riportate in [50], si afferma che quando si stabiliscono tassi di migrazione troppo alti o troppo bassi, l'algoritmo richiede un maggior numero di generazioni per trovare l'ottimo.

In [10] Cohoon, Hedge, Martin e Richards hanno proposto una implementazione parallela dell'algoritmo genetico basata sulla teoria del **punto di equilibrio**. Secondo tale teoria, i processi che evolvono in parallelo sono caratterizzati da due principi:

- **speciaziòne**;

```
Program AG_Parallelo;
begin
popolazione =GENERA(pop_size);
for gen = 1 to num_gens do
begin

        popolazione_fitness=CALCOLA_FITNESS(popolazione);
        if (gen mod frequenza_di_scambio= 0) then
        begin

                individui_migliori=SELEZIONA_INDIVIDUI_MIGLIORI(popolazione);
                INVIA_AI_PROCESSORI_VICINI(individui_migliori);
                RICEVI_DAI_PROCESSORI_VICINI(individui_migliori);
                individui_peggiori=SELEZIONA_INDIVIDUI_PEGGIORI(popolazione);
                RIMPIAZZA(individui_migliori,individui_peggiori);

        end;
        for i = 1 to (pop_size/2) do
        begin

                primo_genitore=SELEZIONA(popolazione);
                secondo_genitore=SELEZIONA(popolazione);
                primo_figlio=CROSSOVER(primo_genitore,secondo_genitore);
                secondo_figlio=CROSSOVER(secondo_genitore,primo_genitore);
                /* Il crossover produce dai medesimi genitori due figli differenti */
                MUTAZIONE(primo_figlio);
                MUTAZIONE(secondo_figlio);

        end;
        INSERISCI_MIGLIOR_INDIVIDUO(popolazione);
        /* Se l'individuo di migliore valore di fitness nella precedente popolazione non è presente nella nuova
        popolazione allora lo si inserisce, togliendo un individuo da quest'ultima */

end;
end
```

Figura 2.9: Algoritmo genetico parallelo di Tanese.

- **stasi**.

Il primo indica una rapida evoluzione di nuove specie (nuovi insiemi di soluzioni) allorchè un insieme di individui, rimasto isolato per qualche generazione, venga a contatto con nuovi individui.

Il secondo indica la mancanza di cambiamenti apprezzabili da una generazione all'altra.

Gli autori creano diversi ambienti (insiemi di soluzioni) che, una volta raggiunta la stasi, vengono perturbati con l'introduzione di individui provenienti da altre sottopopolazioni. Nell'implementazione si suppone che le sottopopolazioni raggiungano la stasi dopo un certo numero di generazioni (chiamate epoche dagli autori), dopo di che avviene la perturbazione che si effettua copiando insiemi casuali di individui nelle sottopopolazioni. Infine, un passo di selezione riconduce le sottopopolazioni alle dimensioni iniziali.

Una caratteristica dell'algoritmo è costituita dall'uso di quattro operatori di *crossover* diversi, applicati non deterministicamente. Gli autori applicano l'algoritmo ad un problema di posizionamento di circuiti. Essi notano che per non influire sulla *performance* dell'algoritmo la topologia di interconnessione deve assicurare alta connettività e basso diametro. Sostengono poi che l'algoritmo parallelo ha migliori *performance* sia dell'algoritmo genetico senza migrazione che dell'algoritmo seriale.

Pettey, Leuze e Grefenstette in [47] hanno implementato un algoritmo parallelo a grana grossa nel quale ad ogni generazione si selezionano gli individui migliori di ogni sottopopolazione, i quali vengono poi trasferiti a tutte le sottopopolazioni vicine. Gli autori riconoscono come il loro algoritmo possa essere considerato molto simile ad un algoritmo panmitico; l'unica differenza è nella selezione che, nell'algoritmo parallelo, avviene a livello di sottopopolazione.

Muhlenbein ed altri [43] hanno implementato un modello di algoritmo genetico a grana grossa per la risoluzione di problemi di ottimizzazione numerica. La tecnica usata prevede l'uso di un ottimizzatore locale all'interno di ogni sottopopolazione, il quale entra in azione cercando di migliorare la qualità delle soluzioni (tramite una ricerca locale) ogni qualvolta si verifichi la mancanza di miglioramenti dopo un certo numero di generazioni. Inoltre la migliore soluzione trovata dalla sottopopolazione viene diffusa alle popolazioni vicine. L'autore usa due misure per rilevare le $performance$ dell'algoritmo: il numero di volte che viene valutata la funzione obiettivo e il tempo richiesto per la computazione. Il primo dato misura la "intelligenza" dell'algoritmo, il secondo la velocità elaborativa. Anche in questo lavoro la migrazione avviene ad intervalli di tempo regolari.

2.1.5 Algoritmi genetici centralizzati

Bianchini e Brown in [6] hanno effettuato una serie di implementazioni di algoritmi genetici per elaboratori a memoria distribuita, comprendenti vari gradi di centralizzazione. Le alternative esaminate spaziano dalla singola popolazione (massima centralizzazione) ad un insieme di sottopopolazioni separate (soluzione completamente distribuita).

Gli autori utilizzano il modello di parallelismo $Master-Slave$. Dal punto di vista logico il modello prevede un processo $master$ ed un certo numero di processi $slave$. Il funzionamento degli $slave$ è coordinato dal $master$ nel modo seguente: i dati in ingresso vengono letti dal $master$ che decide, in base alle informazioni di stato degli $slave$, a quali di questi assegnare il lavoro, cioè gli insiemi di soluzioni da elaborare. Inoltre il $master$ riceve i risultati delle elaborazioni effettuate dagli $slave$. Il flusso di controllo di questi ultimi è generalmente costituito da un ciclo: attesa dati, elaborazione dati, invio risultati al *master* . La fine del ciclo si ha quando il *master* comunica a tutti i *worker* che non ci sono più dati da elaborare.

Gli autori sostengono che le soluzioni basate su un più alto grado di centralizzazione, data la maggior conoscenza sull'evoluzione globale dell'algoritmo, permettono di distribuire a tutti i processori gli individui migliori, aumentando così la velocità di convergenza dell'algoritmo, al prezzo di un maggior costo di comunicazione. Soluzioni più distribuite spendono molto tempo nella ricerca di soluzioni che, pur essendo le migliori a livello locale, possono non risultare globalmente le migliori. La scarsa conoscenza del processo globale delle soluzioni distribuite porta alla perdita di cicli macchina in operazioni poco fruttuose, quale la valutazione di soluzioni non globalmente buone, e pertanto può intrappolare la ricerca delle soluzioni in ottimi locali. Nondimeno, le soluzioni distribuite hanno il vantaggio di possedere un maggior numero di ambienti diversi rispetto alla soluzione centralizzata, ognuno dei quali corrisponde ad una diversa sottopopolazione, e possono quindi esplorare meglio lo spazio delle soluzioni.

Nel Paragrafo seguente esaminiamo due delle implementazioni presentate nel lavoro.

Implementazione centralizzata

Questa implementazione utilizza il modello di parallelismo $Master - Slave$. La topologia di interconnessione è a stella, con il *master* al centro (vedi Figura 2.10).

In una organizzazione completamente centralizzata, il nodo *master* memorizza la popolazione, esegue l'algoritmo di replicazione delle soluzioni[2],

[2] Con il termine replicazione delle soluzioni si intende l'applicazione di una selezione sulla popolazione e quindi degli operatori genetici di *crossover* e mutazione sulla popolazione selezionata.

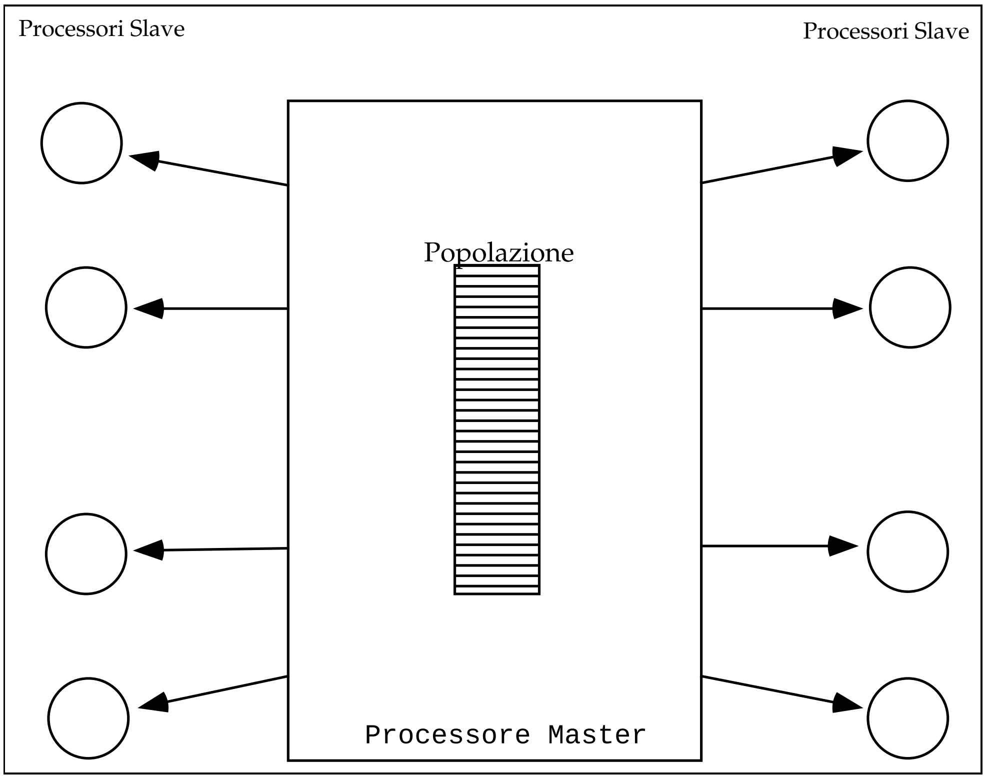


Figura 2.10: Topologia di interconnessione a stella che utilizza il modello di parallelismo $Master - Slave$.

e manda copie degli individui ai processori *slave*. I nodi *slave* ricevono le sottopopolazioni, le trasformano usando gli operatori genetici, verificano la validità delle soluzioni e calcolano il valore di $fitness$ per gli individui generati. Infine restituiscono i nuovi individui (e le informazioni a loro associate) al nodo *master*, il quale esegue di nuovo l'algoritmo di replicazione delle soluzioni. Il processo continua fino al completamento dell'algoritmo.

Il modello di parallelismo $Master - Slave$ non risulta efficiente su architetture distribuite. Il nodo *master* all'aumentare del numero di *slave*, può costituire un **collo di bottiglia** per le comunicazioni, la fase di replicazione delle soluzioni è una computazione sequenziale; infine se la granularità delle sottopopolazioni è troppo fine, si generano troppe comunicazioni. Gli auto-

```
Program MASTER;
begin
INIZIALIZZA_POPOLAZIONE(popolazione);
CALCOLA_FITNESS(popolazione);
while (condizione_di_terminazione) do
begin

        SELEZIONE(sotto_popolazione);
        CROSSOVER(sotto_popolazione);
        MUTAZIONE(sotto_popolazione);
        for j = 1 to num_slave do INVIA_AL_NODO_SLAVE(j,sotto_popolazione);
        for i = 1 to num_gen do
        begin

                SELEZIONE(sotto_popolazione);
                CROSSOVER(sotto_popolazione);
                MUTAZIONE(sotto_popolazione);
                CALCOLA_FITNESS(sotto_popolazione);

        end;
        RICEVI_DAI_NODI_SLAVE(sotto_popolazione);

end;
end
```

Figura 2.11: Struttura del programma *master* per AG parallelo centralizzato di Bianchini e Brown.

ri considerano questi problemi, e ne riducono gli effetti assegnando ai nodi *slave* gruppi più ampi di soluzioni. In Figura 2.11 ed in Figura 2.14 sono schematizzati rispettivamente gli algoritmi dei programmi *master* e *slave*.

Implementazione semi-distribuita

La soluzione semi-distribuita permette di raggiungere un compromesso tra la conoscenza globale della popolazione attuale e la sua distribuzione. Questa implementazione intende rimuovere il *collo di bottiglia* per la *performance* causato dalla presenza del nodo *master* nell'algoritmo centralizzato. L'idea consiste nel raggruppare insiemi di nodi che operano come nell'algoritmo centralizzato. I nodi *master* possono essere interconnessi tramite una struttura ad anello (Figura 2.12). Le comunicazioni tra i nodi *master* per lo scambio delle soluzioni buone possono avvenire con la frequenza desiderata.

Questa organizzazione riduce i conflitti per l'accesso ad un unico no-

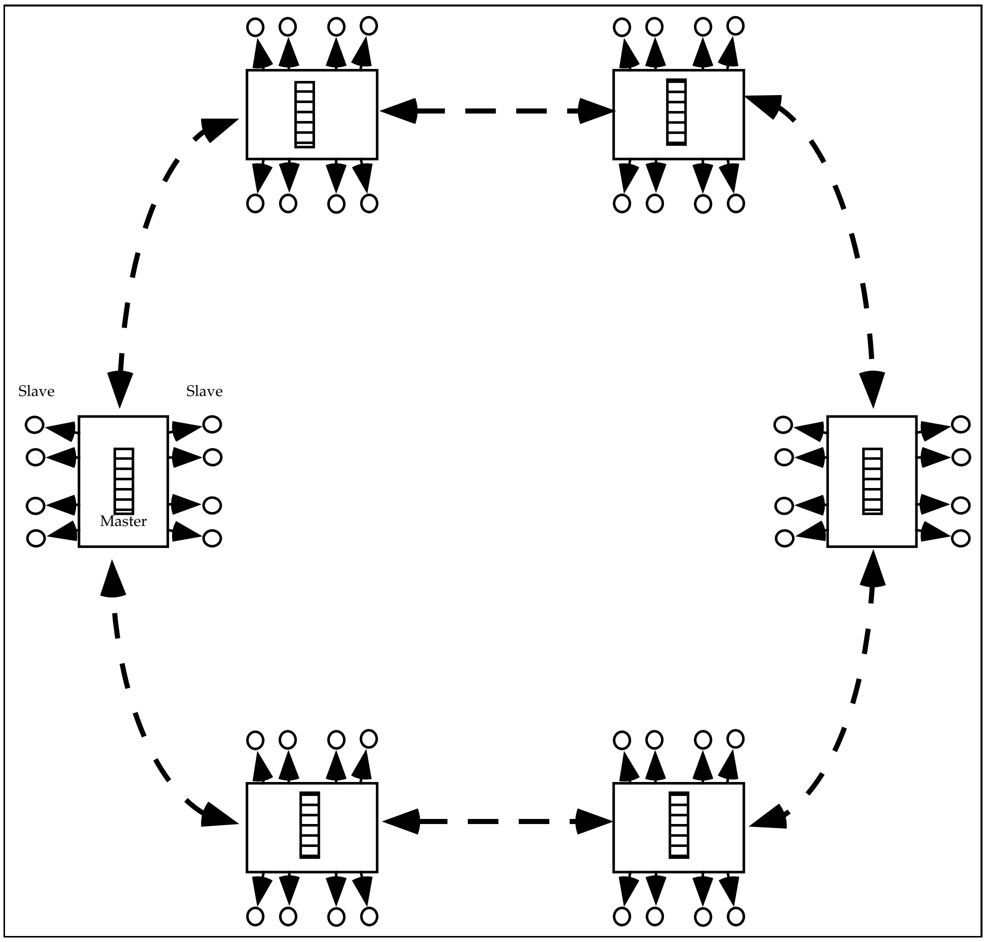


Figura 2.12: Struttura di interconnessione dei nodi *master* nella implementazione semi-distribuita di Bianchini e Brown.

do *master*, a fronte di una riduzione del livello di centralizzazione della popolazione.

In Figura 2.13 ed in Figura 2.14 sono schematizzati rispettivamente gli algoritmi dei programmi *master* e *slave*.

2.2 Conclusioni

I modelli di parallelizzazione a *grana fine*, a *grana grossa* e *centralizzata* illustrati in questo capitolo presentano alcune peculiarità che qui riassumia-

```
Program MASTER;
begin
INIZIALIZZA_POPOLAZIONE(popolazione);
CALCOLA_FITNESS(popolazione);
while (condizione_di_terminazione) do
begin

        SELEZIONE(sotto_popolazione);
        CROSSOVER(sotto_popolazione);
        MUTAZIONE(sotto_popolazione);
        for j = 1 to num_slave do INVIA_AL_NODO_SLAVE(j,sotto_popolazione);
        if (tempo_di_scambio) then
        begin

                migliori=SELEZIONA_INDIVIDUI_MIGLIORI;
                INVIA_AI_NODI_VICINI(migliori);
                RICEVI_DAI_NODI_VICINI(migliori);
                RIMPIAZZA(sotto_popolazione,migliori);

        end;
        for i= 1 to num_gen do
        begin

                SELEZIONE(sotto_popolazione);
                CROSSOVER(sotto_popolazione);
                MUTAZIONE(sotto_popolazione);
                CALCOLA_FITNESS(sotto_popolazione);

        end;
        RICEVI_DAI_NODI_SLAVE(sotto_popolazione);

end;
VISUALIZZA_MIGLIOR_INDIVIDUO(popolazione);
end
```

Figura 2.13: Struttura del programma *master* per AG parallelo semi-distribuito di Bianchini e Brown.

```
Program SLAVE;
begin
while (condizione_di_terminazione) do
begin

        RICEVI_POPOLAZIONE_DAL_MASTER(sotto_popolazione);
        for i = 1 to num_gen do
        begin

                SELEZIONE(sotto_popolazione);
                CROSSOVER(sotto_popolazione);
                MUTAZIONE(sotto_popolazione);
                CALCOLA_FITNESS(sotto_popolazione);

        end;
        MANDA_AL_NODO_MASTER(sotto_popolazione);

end;
end
```

Figura 2.14: Struttura del programma *slave* per AG parallelo di Bianchini e Brown mo.

Il modello a *grana fine* ha la massima decentralizzazione: le decisioni avvengono in maniera esclusivamente locale, ed è quindi adatto per essere eseguito su calcolatori a parallelismo massiccio.

Il modello a *grana grossa* consente una maggiore differenziazione della popolazione rispetto al modello *panmitico* e a quello *a grana fine*; ciò si riflette in una maggiore accuratezza nella ricerca di soluzioni perchè le isole esplorano nello spazio delle soluzioni regioni tra di loro distanti. Il tasso di migrazione degli individui tra le isole è un fattore critico, che può influenzare il buon funzionamento dell'algoritmo.

Il modello *centralizzato* consente un maggiore controllo sull'andamento del processo evolutivo, a costo però di un rallentamento dovuto ai conflitti per l'invio delle informazioni al nodo *master*.

Tutti gli algoritmi paralleli conseguono soluzioni di bontà paragonabile agli algoritmi sequenziali, con il vantaggio di uno *speedup* che, nel caso dei modelli a *grana fine* e *grossa*, si avvicina alla linearità.

Capitolo 3

Algoritmi di simulated annealing

L'algoritmo di **simulated annealing** (indicato con SA nel seguito) costituisce una tecnica generale di ottimizzazione per la risoluzione di problemi di ottimizzazione combinatoriale [2]. L'algoritmo si basa su tecniche stocastiche. Inoltre include un certo numero di concetti che sono in relazione con gli algoritmi di miglioramento iterativo. Dal momento che questi concetti giocano un ruolo fondamentale nella comprensione dell'algoritmo di simulated annealing, tratteremo brevemente le tecniche di ricerca e miglioramento iterativo delle soluzioni.

Gli algoritmi di ricerca e miglioramento iterativo delle soluzioni sono conosciuti col nome di **algoritmi di ricerca locale** [2]. Il funzionamento di questi ultimi si può sintetizzare nel seguente modo: a partire da una configurazione A, si esegue una sequenza di iterazioni, ognuna delle quali consiste di una transizione dalla configurazione corrente verso un'altra che appartiene all'intorno di A. Se la transizione dà luogo ad un miglioramento del costo, la configurazione corrente viene rimpiazzata dalla sua vicina, altrimenti una nuova configurazione tra quelle vicine viene selezionata per un nuovo confronto. L'algoritmo termina quando si ottiene una configurazione che ha costo non peggiore di ogni altra configurazione vicina.

Gli svantaggi di tale algoritmo possono essere così riassunti:

- per definizione, l'algoritmo di ricerca locale termina in un minimo locale, e non si ha nessuna informazione su quanto tale minimo differisca dal minimo globale;
- la qualità del minimo locale ottenuto dipende dalla configurazione iniziale scelta e nessun criterio generale stabilisce un modo per selezionare un punto di partenza dal quale ottenere buone soluzioni;
- in generale, non si possono dare limiti al tempo di completamento dell'algoritmo.

La ricerca locale ha però il vantaggio di essere in generale applicabile con facilità: la definizione delle configurazioni, della funzione costo e del meccanismo di generazione non creano in genere difficoltà. Per rimediare agli svantaggi menzionati, sono state introdotte alcune varianti nell'algoritmo:

1. esecuzione dell'algoritmo su un elevato numero di configurazioni iniziali distribuite uniformemente sull'insieme delle configurazioni possibili;
2. uso delle informazioni ricavate dalle precedenti esecuzioni per migliorare la scelta della configurazione iniziale per la successiva computazione;
3. introduzione di un meccanismo di generazione più complesso (oppure, equivalentemente, un allargamento del concetto di vicinanza), per rendere l'algoritmo capace di sorpassare gli ottimi locali;
4. introduzione della possibilità di accettare configurazioni che forniscono un peggioramento del valore dalla funzione costo. Nell'algoritmo di ricerca locale sono accettati solo miglioramenti del costo delle soluzioni.

Le varianti corrispondenti al secondo e al terzo punto dipendono dal problema trattato. Il primo punto è un modo tradizionale di risolvere con algoritmi di approssimazione problemi di ottimizzazione combinatoriale. Un algoritmo che utilizza il quarto punto è l'algoritmo di SA. Le soluzioni ottenute con l'algoritmo di approssimazione basato sul SA non dipendono dalla configurazione iniziale, e riescono generalmente ad approssimare bene l'ottimo globale. È stato dimostrato teoricamente in [2] che il SA, visto come un algoritmo di ottimizzazione, trova asintoticamente l'ottimo globale. L'algoritmo SA non manifesta gli svantaggi dell'algoritmo di ricerca locale, ed è ugualmente un algoritmo applicabile alla generalità dei problemi.

3.1 Descrizione dell'algoritmo

Nella sua forma originaria [27] l'algoritmo di SA è basato sull'analogia tra il processo di solidificazione dei solidi e il problema della risoluzione di problemi combinatori complessi.

In fisica, il termine **annealing** denota un processo fisico nel quale un solido viene riscaldato progressivamente fino ad una temperatura alla quale raggiunge lo stato liquido, per poi farlo raffreddare lentamente. In questo modo, se la temperatura raggiunta è abbastanza alta e il processo di raffredddamento sufficientemente lento, tutte le molecole si dispongono in una configurazione di minima energia. A partire dalla massima temperatura raggiunta, il processo può essere descritto come segue. Per ogni valore T della temperatura il solido raggiunge l'equilibrio termico, caratterizzato da una probabilità di essere in uno stato di energia E data dalla **distribuzione di Boltzmann** :

$$Pr(\mathbf{E} = E) = \frac{1}{Z(T)} \cdot e^{(\frac{-E}{k_B \cdot T})}$$

dove Z(T) è un fattore di normalizzazione che dipende dalla temperatura T, e k_B è la *costante di Boltzmann*. Il fattore $e^{\frac{-E}{k_B \cdot T}}$ è detto *fattore di Boltzmann*. Al decrescere della temperatura, la *distribuzione di Boltzmann* fornisce un numero crescente di stati con energia più bassa; quando la temperatura si avvicina a zero, l'unico stato energetico che ha probabilità di verificarsi è quello di energia minima.

È noto da [27] che se il raffreddamento è troppo rapido, per esempio se non si permette al solido di raggiungere l'equilibrio termico per ogni valore di temperatura, alcuni difetti possono essere inclusi nella struttura del solido, arrivando così a formare strutture amorfe. L'*annealing* deve invece conseguire la formazione di una struttura regolare di minima energia. Per simulare l'evoluzione dell'equilibrio termico di un solido per un valore fissato T di temperatura, Metropolis e altri [37] hanno proposto il metodo **Monte Carlo**.

Quest'ultimo genera sequenze di stati nel modo seguente: dato l'attuale stato del solido, caratterizzato dalle posizioni delle sue molecole, viene applicata una piccola perturbazione casuale ad un sottoinsieme delle particelle. La differenza di energia, $\Delta(E) = E_i - E_j$, tra lo stato corrente i e quello perturbato j può risultare:

- positiva: si è trovato uno stato che abbassa l'energia. Allora il nuovo stato con energia minore rimpiazza lo stato corrente;
- negativa: il nuovo stato, che aumenta l'energia del solido, viene accettato con probabilità $e^{\frac{-E}{k_B \cdot T}}$.

Quello appena illustrato è il **Criterio di Metropolis**.

Seguendo questo criterio, per un sistema che evolve fino al raggiungimento dell'equilibrio termico, la distribuzione delle probabilità degli stati segue la *distribuzione di Boltzmann*.

L'algoritmo di Metropolis può essere usato per generare sequenze di configurazioni di un problema di ottimizzazione combinatoriale. In tal caso, le configurazioni assumono il ruolo degli stati del solido, mentre l'energia e la temperatura vengono rimpiazzati, rispettivamente, dalla funzione costo C e dal parametro di controllo c.

L'algoritmo di SA può essere quindi visto come una sequenza di esecuzioni dell'algoritmo di Metropolis, valutati con una successione decrescente di valori del parametro di controllo.

L'algoritmo di SA può essere così sintetizzato: inizialmente si dà un alto valore al parametro di controllo, quindi si genera una sequenza di configurazioni, fino ad arrivare al punto di minimo costo. Le configurazioni vengono scelte tra quelle presenti nell'**intorno**[1] della soluzione corrente, secondo il seguente criterio: se $(\Delta(Costo) < 0)$[2] allora la nuova configurazione rimpiazza la vecchia, altrimenti se $(\Delta(Costo) \geq 0)$ la probabilità di rimpiazzare la vecchia configurazione con la nuova è data da $e^{\frac{\Delta Costo}{k_B \cdot T}}$. Quindi c'è una probabilità non negativa di accettare soluzioni di costo più alto. Il processo continua fino a che la distribuzione della probabilità delle configurazioni si avvicina alla *distribuzione di Boltzmann*.

Possiamo scrivere la probabilità con la quale ogni configurazione può presentarsi nel modo seguente:

$$Pr(\mathbf{configurazione_i}) = \frac{1}{Q(c)} \cdot e^{\frac{\Delta Costo}{c}}$$

dove $Q(c)$ è una costante di normalizzazione che dipende dal parametro di

[1] Le configurazioni presenti nell'intorno della configurazione corrente differiscono il minimo possibile da quest'ultima.

[2] Dove $\Delta(Costo) = nuovo\_costo - vecchio\_costo$, è la differenza tra il costo della nuova e la vecchia configurazione

controllo c. Il valore del parametro di controllo è abbassato progressivamente a passi discreti, ed al sistema viene permesso ad ogni passo di raggiungere l'equilibrio, nel modo descritto sopra. L'algoritmo termina per un valore di c, oltre il quale nessuna configurazione che peggiori il valore della funzione costo è accettata. La soluzione così ottenuta è la soluzione del problema trattato.

Possiamo ora descrivere con maggiore precisione l'algoritmo di approssimazione basato su SA. Definite le configurazioni, la funzione costo e un meccanismo di generazione delle configurazioni successive, un problema di ottimizzazione combinatoriale può essere risolto dall'algoritmo schematizzato in Figura 3.1.

Si noti che il criterio di accettazione di nuove configurazioni sopra illustrato viene implementato nell'algoritmo generando numeri casuali con una distribuzione di probabilità uniforme sull'intervallo [0,1], e confrontando tali valori casuali con $e^{\frac{\Delta(Costo)}{c}}$. La probabiltà di accettare soluzioni che aumentino il costo della configurazione diminuisce al decrescere della temperatura.

Confrontando gli algoritmi di ricerca locale con l'algoritmo basato su SA si rilevano le seguenti analogie e differenze:

- un algoritmo basato su SA con il parametro di controllo fissato a 0 fin dall'inizio del processo opera in modo analogo ad un algoritmo di ricerca locale;

- un algoritmo di ricerca locale può scegliere in maniera non casuale le configurazioni appartenenti all'intorno della soluzione corrente, mentre nell'algoritmo SA tale scelta è casuale;

- l'algoritmo basato su SA è una generalizzazione della ricerca locale,

```
Program Simulated_Annealing;
begin
c = c0;
configurazione_corrente= INIZIALIZZA();
costo = CALCOLA_COSTO(configurazione_corrente);
repeat

        repeat

                nuova_configurazione = PERTURBA(configurazione_corrente);
                /*Ottiene una nuova configurazione modificando configurazione_corrente*/
                nuovo_costo = CALCOLA_COSTO(nuova_configurazione);
                Δ(Costo) = nuovo_costo − costo;
                if ((Δ(Costo) < 0) ⋁ (e^(Δ(Costo)/c)) > random(0,1)) then
                begin

                        configurazione_corrente= nuova_configurazione;
                        costo = nuovo_costo;
                        /* Se la nuova configurazione ha costo migliore, oppure se
                        il criterio di accettazione stabilisce l'accettabilità della nuova
                        configurazione, nuova_configurazione diventa la configurazione
                        corrente */


                end;

        until (EQUILIBRIO());
        /* Il ciclo continua fino a ristabilire le condizioni di equilibrio */
        c = f(c);

until ( c < cf );
end
```

Figura 3.1: Algoritmo di Simulated Annealing (c_0 e c_f sono i valori iniziale e finale del parametro c).

nella quale si accettano soluzioni peggiorative della funzione costo con probabilità diversa da zero;

- l'algoritmo basato sulla tecnica SA è indipendente dal problema trattato.

3.2 Scelta dei parametri nel SA

Le prestazioni dell'algoritmo di approssimazione basato sul SA possono variare al mutare di alcuni parametri implementativi, che sono:

- il valore iniziale del parametro di controllo (c_0);
- il valore finale del parametro di controllo (c_f), che definisce il **criterio di arresto** dell'algoritmo;
- la regola che stabilisce come far variare il parametro di controllo c_k dall'iterazione corrente k, alla successiva $(k+1)$, detto **criterio di decremento**.

I valori specificati per questi parametri definiscono il **programma di raffreddamento** dell'algoritmo basato su SA (l'analogia è sempre con il processo di raffreddamento dei solidi).

Un *criterio di arresto* può far terminare l'esecuzione dell'algoritmo quando il miglioramento delle soluzioni diventa trascurabile.

Nello stabilire il *criterio di decremento* del parametro di controllo c, si deve tener presente che abbassando troppo velocemente il valore di c, si richiedono molte iterazioni prima di raggiungere di nuovo lo stato di equilibrio. Al crescere del numero di iterazioni necessarie a ristabilire l'equilibrio, cresce anche il tempo necessario a completare l'esecuzione dell'algoritmo.

3.2.1 Valore iniziale del parametro di controllo

Il valore iniziale di c viene fissato in modo che tutte le transizioni siano accettate ($e^{\frac{\Delta Costo)}{c_0}} \approx 1$). Kirkpatrick ha proposto in [27] la seguente regola empirica: si scelga un valore abbastanza grande di c_0, e si esegua l'algoritmo per un certo numero di volte. Se il **tasso di accettazione** X (definito come il numero di transizioni accettate diviso per il numero di transizioni proposte) è minore del valore iniziale X_0 (in [27] X_0 è fissato al valore 0.8), si raddoppia il valore attuale di c. Si continua la procedura finchè il *tasso di accettazione* osservato supera X_0.

Questa regola è stata raffinata da vari autori: Johnson in [25] ha, ad esempio, determinato c_0 calcolando la media dell'incremento dei costi $M_{\Delta(C)}$ per un numero di transizioni casuali.

Dalla formula:

$$X_0 = e^{\frac{-M_{\Delta(C)}}{c_0}}$$

si ricava c_0:

$$c_0 = \frac{M_{\Delta(C)}}{ln(X_0)^{-1}}$$

3.2.2 Valore finale del parametro di controllo

Un *criterio di arresto* può essere determinato nei seguenti modi:

- fissando il numero di possibili valori del parametro di controllo c per i quali l'algoritmo viene eseguito;
- stabilendo che dopo la ripetizione consecutiva di un certo numero di configurazioni identiche, l'esecuzione dell'algoritmo termina.

3.2.3 Valore di decremento del parametro di controllo

Come detto sopra, il decremento deve essere tale da permettere il ristabilimento della situazione di equilibrio. Una regola di decremento di uso comune è la seguente:

$$c_{k+1} = ac_k$$

con k che assume valori interi positivi. In [27] viene proposto un valore del parametro a pari a 0.95 .

3.3 Algoritmi ibridi: SA combinato con la ricerca locale

In [35] viene proposta una implementazione dell'algoritmo di SA che usa una ricerca locale e un meccanismo chiamato **salto**, usato per esplorare meglio lo spazio delle soluzioni. Gli autori mettono in risalto come l'algoritmo di SA possa essere migliorato introducendo una strategia che campioni le soluzioni scegliendole solo tra gli ottimi locali. Per fare ciò bisogna realizzare una strategia che consenta di andare da un ottimo locale all'altro, oltrepassando tutte le configurazioni che stanno tra i due ottimi, la cui valutazione comporterebbe un aumento del tempo di esecuzione dell'algoritmo. Questa strategia viene realizzata dal meccanismo del *salto* e dalla successiva applicazione di una ricerca locale. Successivamente un algoritmo di ricerca locale si occupa di trovare un ottimo locale.

Un esempio di uso di detta strategia di *salto* nella valutazione di una funzione obiettivo è mostrato in Figura 3.2.

Partendo dalla configurazione A, il *salto* porta alla configurazione B; successivamente la ricerca locale fa assestare la soluzione corrente nel punto C.

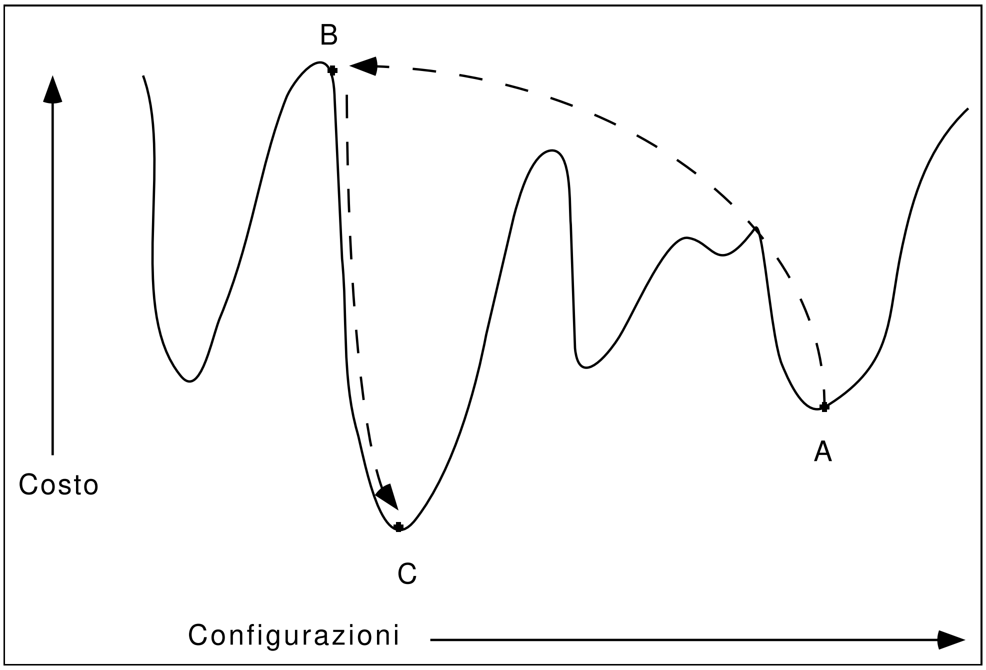


Figura 3.2: Esempio di meccanismo di *salto* usata nell'algoritmo di Martin e Otto per la valutazione di una funzione obiettivo.

Quindi si accetta o si rifiuta la nuova soluzione secondo la normale procedura dell'algoritmo di SA. Senza questa strategia, l'algoritmo di SA avrebbe esplorato tutto lo spazio delle soluzioni, e per giungere al punto C avrebbe dovuto effettuare una serie di accettazioni e rifiuti.

L'algoritmo di ricerca locale e il meccanismo di *salto* dipendono dal problema trattato. Nel caso del problema del commesso viaggiatore (TSP), che costituisce il caso di studio del lavoro esposto in [35], gli autori impiegano l'algoritmo di ricerca locale di Lin-Kerninghan [31].

La scelta del *salto* porta gli autori a cercare una trasformazione della soluzione che non sia possibile conseguire attraverso la semplice applicazione dell'algorimo di ricerca locale. Dal momento che l'algoritmo di Lin-Kerninghan cerca di migliorare la soluzione dapprima applicando tutte le trasformazioni che scambiano 2 archi del percorso, per poi passare alle trasformazioni su

3 archi, e così via, gli autori implementano il *salto* applicando una trasformazione su 4 archi, che non può essere ottenuta come composizione di due trasformazioni successive su 2 archi. Per fare ciò dapprima effettuano una trasformazione su 2 archi che sconnette il percorso, poi lo riconnettono con un'altra trasformazione di 2 archi. Un esempio di *salto* è riportato in Figura 3.3; si possono vedere gli archi che connettono le città (che non sono rappresentate), e due possibili percorsi per connettere tutte le città. Il primo percorso è quello precedente al *salto*, e si ottiene considerando solamente gli archi disegnati con la linea non tratteggiata; il secondo percorso (successivo al *salto*) si ottiene togliendo gli archi contraddistinti con le lettere A, B, C, D e inserendo al loro posto i quattro archi tratteggiati. I test effettuati dimostrano come l'algoritmo si comporti meglio del metodo SA standard, riuscendo a trovare la soluzione ottima del problema del commesso viaggiatore con 783 città in circa un'ora. Il limite della strategia è che la definizione del *salto* dipende fortemente dal problema trattato.

3.4 Conclusioni

In questo capitolo abbiamo presentato l'algoritmo di SA e introdotto brevemente gli algoritmi di ricerca locale. Entrambi gli algoritmi si adattano facilmente ai vari problemi di ottimizzazione combinatoriale. L'algoritmo di SA presenta alcuni vantaggi sull'algoritmo di ricerca locale:

- la qualità della soluzione trovata non dipende, come nell'algoritmo di ricerca locale, dalla soluzione di partenza;
- è stata dimostrata la convergenza in un tempo finito dell'algoritmo di SA verso l'ottimo globale.

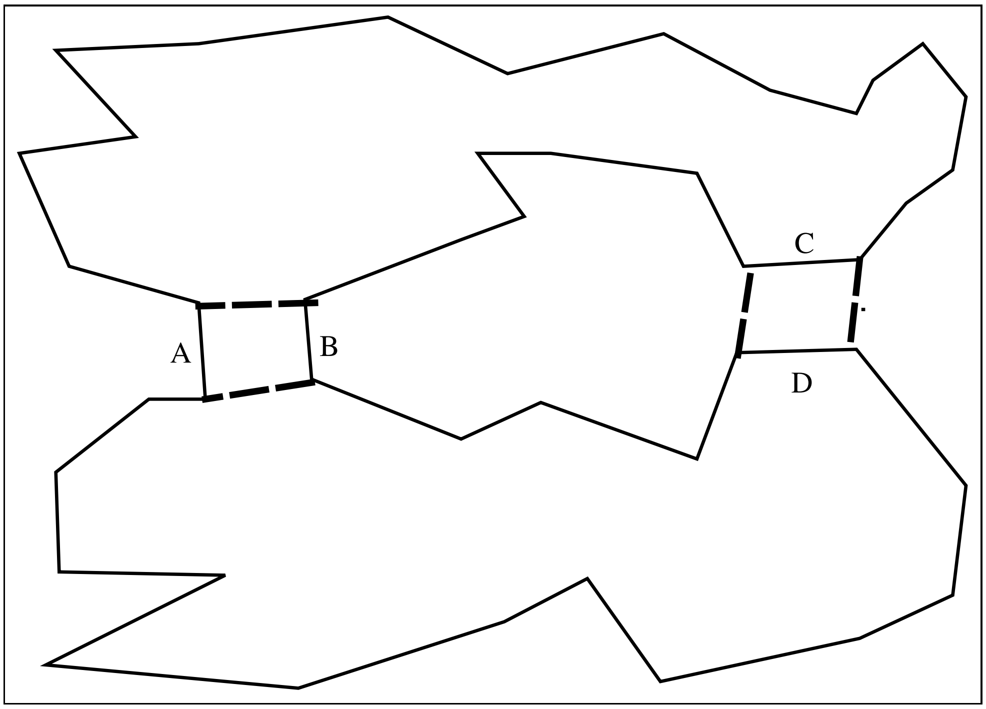


Figura 3.3: Esempio di *salto* per il problema TSP usato da Martin e Otto

Naturalmente si hanno anche degli aspetti negativi: l'algoritmo di SA per poter fornire buone soluzioni richiede in genere un lungo tempo di esecuzione se confrontato, ad esempio, con quello necessario a raggiungere la convergenza con un *algoritmo genetico*.

Capitolo 4

Algoritmi paralleli di simulated annealing

Uno dei principali svantaggi del metodo SA è che l'applicazione dell'algoritmo può richiedere molto tempo di esecuzione. Un sostanziale miglioramento di quest'ultimo si può ottenere con implementazioni che possono essere eseguite su elaboratori paralleli.

Una implementazione parallela dell'algoritmo di SA dovrebbe possedere le seguenti caratteristiche:

- essere generale ed applicabile a tutti i problemi;
- avere le stesse proprietà di convergenza all'ottimo (in particolare la dimostrabilità della convergenza stessa) dell'algoritmo di SA sequenziale.

La scrittura di una versione parallela dell'algoritmo di SA non risulta difficile se si violano le due caratteristiche sopra elencate. Infatti la maggior parte delle implementazioni proposte rilasciano il vincolo della strettà sequenzialità di mosse successive e permettono a più processori la contemporanea modifica della configurazione del sistema; mentre solo alcune implementazioni non violano le caratteristiche sopra elencate.

Nel seguito analizziamo alcune implementazioni parallele dell'algoritmo di SA.

4.1 Soluzioni parallele dell'algoritmo di SA

In [29] si applica il SA al problema del posizionamento delle *celle* su un *chip*. L'algoritmo è formulato in modo da poter essere eseguito su macchine multiprocessore a memoria condivisa. Gli autori assumono un modello ideale di SA, nel quale il tempo necessario a generare, valutare e accettare o rifiutare una **mossa**[1] sia costante. Ne segue che l'esecuzione dell'algoritmo consiste in una sequenza di mosse di accettazione o rifiuto delle soluzioni e il suo tempo di esecuzione dipende dalla lunghezza di tale sequenza e dal tempo impiegato a fare una mossa. Gli autori distinguono due approcci al problema della parallelizzazione dell'algoritmo di SA:

- **a decomposizione di mossa**;
- **a mosse parallele**.

Il criterio *a decomposizione di mossa* permette di ridurre il tempo di elaborazione di una singola mossa distribuendo l'elaborazione tra più processori.

Il criterio *a mosse parallele* prevede l'esecuzione concorrente, ad ogni passo dell'algoritmo, di più mosse. Un esempio è riportato nella Figura 4.1.

I due approcci non sono mutuamente esclusivi: i criteri sono applicabili contemporaneamente. Per caratterizzare una strategia di partizionamento dell'algoritmo di SA gli autori definiscono due parametri:

[1] Con il termine *mossa* si indica una nuova soluzione ottenuta variando una soluzione preesistente.

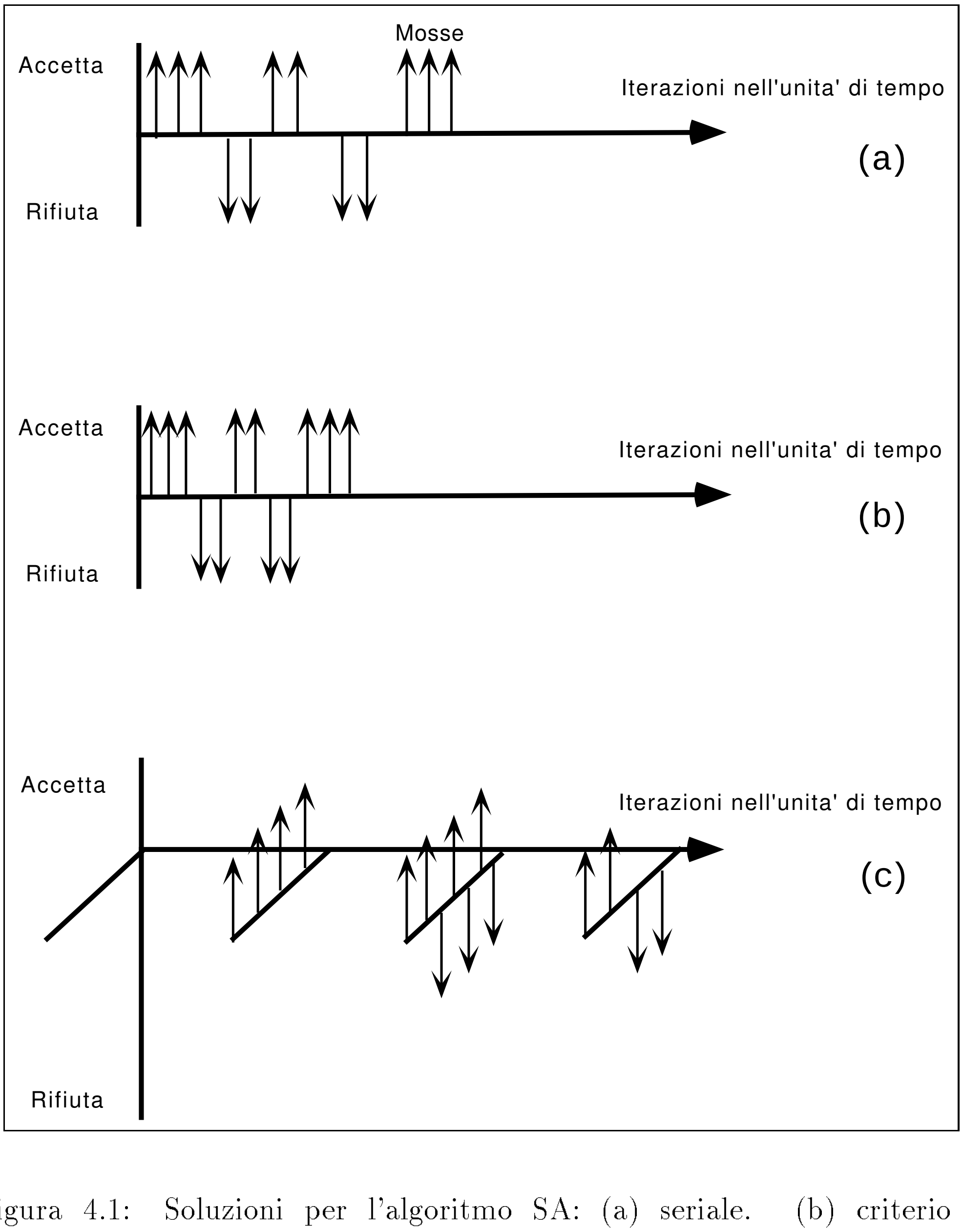


Figura 4.1: Soluzioni per l'algoritmo SA: (a) seriale. (b) criterio *a decomposizione di mossa*. (c) criterio *a mosse parallele*. Dal lavoro di Kravitz e Rutenbar.

- **granularità della mossa**: numero di processi che in parallelo elaborano una singola mossa;
- **parallelismo della mossa**: numero di mosse valutate in parallelo.

La scelta dei parametri di *granularità della mossa* e *parallelismo della mossa* è strettamente dipendente dal costo di comunicazione e di sincronizzazione tra i processori. Le strategie di parallelizzazione sono distinte in due tipi fondamentali:

- **strategie di decomposizione delle mosse**;
- **strategie di parallelizzazione delle mosse**.

La *strategia di decomposizione delle mosse* prevede una decomposizione e una ripartizione in parallelo del lavoro necessario a completare una mossa dell'algoritmo di SA. Una mossa dell'algoritmo di SA consiste nella generazione di una perturbazione ammissibile della soluzione corrente, nella valutazione della variazione del costo, ed infine nell'accettazione o nel rifiuto della soluzione generata. Alcune di queste attività possono essere condotte in parallelo. Si distinguono due tipi di decomposizione delle mosse:

- **decomposizione per oggetto**;
- **decomposizione per funzione**.

Una decomposizione *per oggetto* delega il compito di lavorare su un sottoinsieme di dati ad un solo processore. In tal modo tutte le operazioni su un certo insieme di dati sono compito di un solo processore.

Una decomposizione *per funzione* delega l'esecuzione di certe funzioni ad un solo processore. Tali funzioni saranno quindi eseguite solo dal processore a cui sono state assegnate.

Per esempio, nel problema di posizionamento delle *celle* su un *chip*, una decomposizione *per oggetto* può assegnare un insieme di *celle* ad un unico processore, mentre una decomposizione *per funzione* può assegnare ad un processore il compito di valutare la lunghezza delle connessioni tra le *celle*.

I due tipi di decomposizione si possono ulteriormente suddividere, distinguendo strategie **statiche** e **dinamiche**. Se la distribuzione degli oggetti o delle funzioni ai processori è effettuata una sola volta all'inizio della elaborazione, abbiamo una strategia *statica*.

La strategia *dinamica* prevede, invece, un assegnamento delle funzioni o degli oggetti ai processori che cambia durante l'esecuzione dell'algoritmo.

Una vista schematica delle possibili decomposizioni applicabili all'algoritmo sequenziale si SA è riportato in Figura 4.2.

Le decomposizioni *per oggetto* e *per funzione* hanno lo svantaggio, al crescere del parallelismo, di richiedere un numero crescente di sincronizzazioni, con conseguente aumento dell'*overhead*.

La strategia *statica*, rispetto alla strategia *dinamica*, ha il vantaggio di eliminare il tempo di computazione necessario per il riassegnamento degli oggetti o delle funzioni ai processori; l'esecuzione può soffrire però di uno sbilanciamento del carico.

Il parallelismo della strategia a decomposizione di mosse è limitato dal numero di celle e connessioni tra celle che si trattano contemporaneamente.

I metodi a *strategia di parallelizzazione delle mosse* sono caratterizzati da come i processori scelgono le mosse da valutare. Ogni processore può scegliere indipendentemente da tutti gli altri processori alcune mosse tra quelle possibili. In alternativa, si può stabilire un criterio di scelta delle mosse in base al quale ogni processore scelga le mosse da effettuare. Un esempio di scelta delle mosse per il problema del posizionamento delle *celle* su un *chip* è il seguente: si partiziona l'area totale del *chip* in alcune regioni, e si assegna ad ogni processore il compito di effettuare tutte le mosse che riguardano la regione di sua competenza.

In una implementazione parallela, sorge il problema della interazione delle mosse. Mosse fatte in parallelo possono essere in conflitto: per esempio, non è corretto muovere la stessa cella del circuito in due posti diversi.

Inoltre, durante la valutazione della mossa corrente, un processore non può prevedere le conseguenze di tutti i movimenti effettuati dagli altri processori, quindi deve assumere che le uniche *celle* che cambiano posizione siano quelle coinvolte nella mossa locale al processore stesso. Succede allora che decisioni localmente corrette, che decrementano cioè il costo della soluzione locale, possano risultare globalmente in un incremento della funzione costo.

Queste erronee decisioni possono condurre ad oscillazioni della bontà delle soluzioni trattate, e addirittura non permettere la convergenza dell'algoritmo. Una possibile soluzione è quella di non introdurre alcun controllo che risolva o riduca tali inconvenienti; gli autori sostengono che lo spostamento in parallelo di poche *celle* in *chip* che ne comprendono molte non influenza la convergenza dell'algoritmo. L'alternativa consiste nell'introduzione di controlli finalizzati a proibire l'accettazione in parallelo delle mosse che modificano le stesse *celle*. Ad esempio si possono selezionare un insieme di mosse per ogni processore, assicurandosi che siano valutati in parallelo solo quegli insiemi che operano su *celle* diverse. È anche possibile, dopo la valutazione delle mosse fatte in parallelo, rimediare agli effetti indesiderati.

Nel lavoro [29] sono posti a confronto i due algoritmi che implementano queste strategie.

Gli autori mostrano che la *strategia di parallelizzazione delle mosse* funziona meglio a temperature basse, mentre la *strategia di decomposizione*

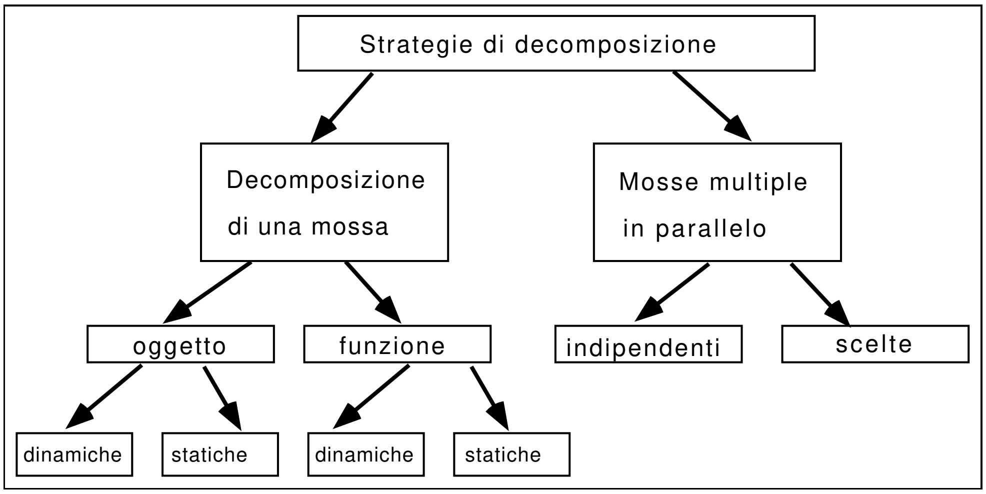


Figura 4.2: Tassonomia di Kravitz e Rutenbar delle decomposizioni applicabili all'algoritmo di SA.

delle mosse è indipendente dalla temperatura, ma è influenzata da come è fatta la partizione delle mosse possibili.

Inoltre, gli autori implementano una terza soluzione che sfrutta la combinazione delle due strategie precedenti. Questa utilizza una decomposizione delle mosse alle alte temperature, e una parallelizzazione delle mosse alle basse temperature. I risultati sperimentali mostrano come questa scelta porti l'algoritmo a *performance* migliori rispetti a quelle dei due algoritmi precedenti.

Una differente implementazione parallela dell'algoritmo di SA applicato al problema del posizionamento delle *celle* si trova in [11]. Il metodo applicato consiste nell'assegnare ad ogni processore una coppia di *celle* scelte a caso nella configurazione corrente. Una volta che una coppia di *celle* è stata allocata ad un processore, questa non può, successivamente, essere assegnata a nessun altro processore (meccanismo di *bloccaggio*). Un processore che trova due *celle* già assegnate, non aspetta che queste si rendano disponibili, ma ne seleziona altre due. Quindi calcola l'effetto che ha sul costo della configurazione una variazione della posizione delle *celle*. La variazione è accettata o rifiutata in base al costo conseguito.

Gli autori hanno implementato due variazioni dell'algoritmo: la prima (indicata nel seguito con A) prevede il bloccaggio delle *celle* e dei loro collegamenti. Con questo metodo, quando una mossa viene accettata, la lunghezza delle connessioni della configurazione corrente è calcolata correttamente. Il numero di processori che possono essere usati è però limitato dalla dimensione del circuito trattato.

Il secondo metodo (indicato nel seguito con B) prevede il bloccaggio delle sole *celle*, rilasciando cioè il vincolo del bloccaggio dei collegamenti. Le mosse simultanee dei processori non permettono però di calcolare correttamente la lunghezza delle connessioni. Per ovviare a questo inconveniente, gli autori hanno proposto una variazione della soluzione B (indicato nel seguito con B1) nella quale la lunghezza delle connessioni viene ricalcolata al termine delle prove effettuate ad ogni temperatura. In questo modo, al termine della computazione di ogni temperatura, si introduce una sincronizzazione tra tutti i processori.

Gli esperimenti condotti mostrano che, dal punto di vista della bontà delle soluzioni, gli algoritmi paralleli si comportano come l'algoritmo sequenziale. Lo *speedup* dei metodi A e B è all'incirca lo stesso per un numero di processori tra due e quattro. Per un maggior numero di processori, il metodo A ottiene una minore efficienza rispetto al B. I metodi B e B1 hanno uno *speedup* molto simile. B1 però trova meno frequentemente buone soluzioni rispetto a B. Gli autori concludono quindi che la soluzione B1 può essere applicata al problema del posizionamento delle *celle* impiegando un numero di queste ultime pari al 10-20% del numero totale di *celle*, ottenendo così una buona efficienza.

Il lavoro proposto in [3] è una versione parallela del SA implementata in OCCAM[2]. La versione sequenziale dell'algoritmo SA viene decomposta dagli autori in due parti:

- una che determina il cambiamento della temperatura e le mosse;
- l'altra che calcola la variazione di energia ΔE associata ad ogni mossa e decide l'accettazione della nuova soluzione.

Gli autori hanno implementato le funzionalità della prima parte mediante un processo, mentre la seconda parte è stata realizzata con diversi processi. In Figura 4.3 è mostrato lo schema logico dell'algoritmo.

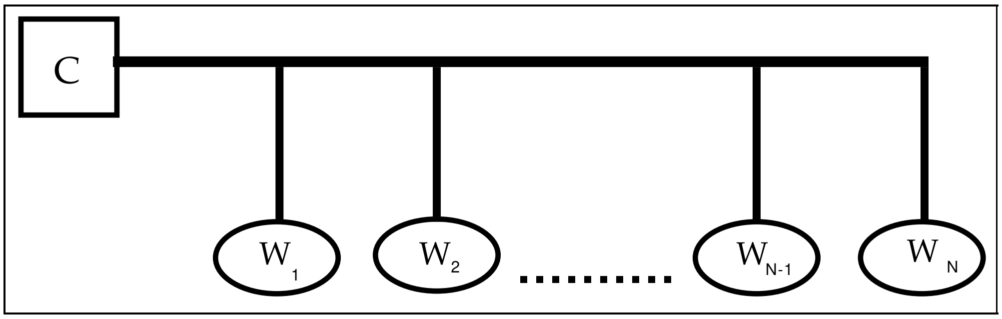


Figura 4.3: Schema logico dell'algoritmo di Baiardi.

Nella figura il processo *master* C realizza le funzionalità incluse nella prima parte, mentre le funzionalità della seconda parte sono realizzate dai processi $W_1....W_n$. Il processo C determina le mosse che costituiscono il pacchetto di lavoro inviato ai processi *slave* W_i e le distribuisce al primo processo W_i libero.

Ogni W_i elabora le mosse e comunica il suo risultato al processo C. Le mosse che vengono accettate sono distribuite a tutti i W_i, poichè ogni W_i usa una copia privata della configurazione.

[2]Linguaggio concorrente che deriva dalla semantica del CSP.

Tale algoritmo parallelo offre i seguenti vantaggi:

- scalabilità;
- dopo aver fissato il numero di W_i, il parallelismo reale può essere controllato dinamicamente attraverso la quantità di dati che C trasferisce ai W_i;
- la strategia di scelta della mossa è incapsulata nel modulo C, che può così facilmente essere rimpiazzato per risolvere problemi diversi.

Gli svantaggi sono:

- le mosse potrebbero essere elaborate simultaneamente da distinti W_i. Se entrambe le mosse sono accettate, quando viene aggiornata la configurazione, è possibile che l'ordine seriale degli aggiornamenti non sia lo stesso su ogni W_i. Perciò è necessario bloccare gli elementi che si devono scambiare di posizione;
- ogni W_i ha una copia privata della configurazione. Questa copia potrebbe differire dalle configurazioni degli altri W_i, almeno finchè W_i non riceve la copia aggiornata dagli altri W_i, con $i \neq j$. Questa inconsistenza potrebbe causare un errore nel calcolare la nuova energia quando W_i deve determinare il risultato della nuova mossa;
- ogni W_i, tra due tentate mosse successive, o durante il calcolo della differenza di energia, potrebbe ricevere una configurazione aggiornata dagli altri processi. L'algoritmo sequenziale calcola l'energia solo quando una mossa è accettata. Nell'implementazione parallela, questa scelta potrebbe causare delle inconsistenze sul valore dell'energia, poichè gli altri processi potrebbero aggiornare la configurazione. Per evitare tale

inconsistenze, ogni W_i calcola sempre l'energia prima di cercare una nuova configurazione.

Gli svantaggi sopra elencati possono essere ridotti mediante una strategia che preveda di bloccare i collegamenti tra le *celle*. Però questo metodo abbassa il grado di parallelismo.

In [9], l'algoritmo di SA viene parallelizzato a livello dei dati. Gli autori, come caso di studio, hanno trattato il problema del piazzamento delle *celle* su un *chip*. Essi hanno adottato la soluzione in cui l'insieme di *celle* C_T che devono essere assegnate ad ogni processore viene partizionato in P sottoinsiemi con P uguale al numero dei processori. La partizione iniziale è selezionata casualmente. Nell'ottica del bilanciamento del carico ad ogni processore viene assegnato lo stesso numero di *celle*.

In tale modello ogni processo esegue solo le mosse sulle *celle* che gli sono state assegnate e la principale sorgente di errore nella valutazione dell'energia è il sovrapporsi di due blocchi che appartengono a due differenti sottoinsiemi. In Figura 4.4 è mostrata tale situazione. Inizialmente le due *celle* x e y sono disposte secondo la configurazione mostrata in Figura 4.4(a); il processo A elabora il blocco x e il processo B elabora il blocco y. Il processo A tenta di muovere la sua cella x a destra, mentre, contemporaneamente, il processo B tenta di muovere la sua cella y a sinistra. Entrambe le configurazioni in Figura 4.4(b) e Figura 4.4(c) sono migliori di quella precedente in Figura 4.4(a) perchè l'area totale del *chip* è diminuita. Da un punto di vista globale si ha un errore perchè si ha un sovrapporsi dei blocchi, come si vede in Figura 4.4(d). Allo scopo di ridurre questo errore, i blocchi sono allocati dinamicamente in modo tale che i blocchi vicini siano assegnati allo stesso processo.

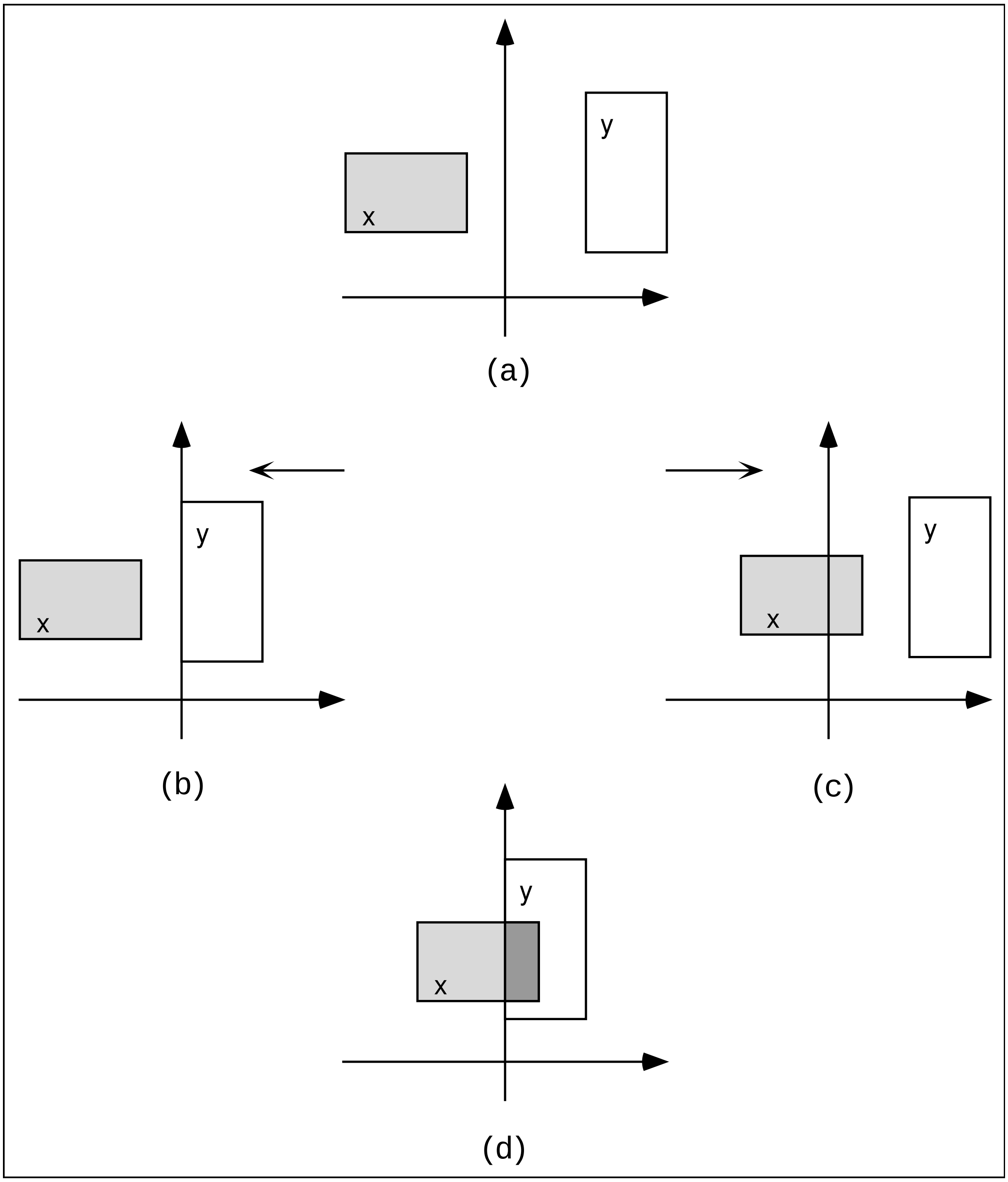


Figura 4.4: Esempio di inconsistenza

In [1] gli autori propongono una soluzione in cui ogni processore esegue un algoritmo di SA sequenziale indipendente dagli altri. Essendo l'algoritmo di SA un algoritmo probabilistico, ci si può aspettare che le diverse istanze dell'algoritmo in esecuzione su processori differenti producano risultati indipendenti. Pertanto, periodicamente, i vari processi si coordinano tra di loro; cioè tutti i processori eseguono un numero di iterazioni fissate, collezionano tutte le configurazioni attuali, scelgono quella di costo minore e usano questa configurazione come nuovo stato iniziale. Tale modo di parallelizzare il SA è stato denominato a *strategia di divisione*.

Gli stessi autori hanno portato delle modifiche alla soluzione precedente. Inizialmente quando la temperatura è alta l'algoritmo modificato si comporta come quello a *strategia di divisione*; successivamente, all'abbassarsi della temperatura l'algoritmo sfrutta il modello $Master - Slave$.

La soluzione $Master - Slave$ prevede che ogni processo *slave* generi una nuova configurazione, calcoli la differenza di energia e decida se accettare o no la nuova configurazione. Quando una nuova configurazione è accettata, i processori *slave* la inviano al processo *master*. Quest'ultimo seleziona la soluzione corrente e la ridistribuisce ai processi *slave*.

4.2 SA parallelo con calcolo speculativo

In [46] l'algoritmo di SA è stato parallelizzato adottando una metodologia che utilizza la tecnica del calcolo speculativo (già proposta in [52]). Usando la tecnica del calcolo speculativo l'algoritmo parallelo di SA verifica le proprietà di mantenere la sequenza decisionale dell'algoritmo sequenziale e di essere indipendente dal problema trattato. Tale algoritmo è stato implementato su un elaboratore $nCube2$ [44]. Esaminando lo pseudocodice riportato in Figura 3.1, possiamo notare che ogni iterazione è composta da tre fasi:

modifica dello stato, valutazione del costo e decisione. L'iterazione corrente dipende dalla soluzione e dal costo calcolati nell'iterazione precedente. Se l'iterazione i si conclude con l'accettazione della nuova configurazione S_i di costo C_i, l'iterazione successiva $i+1$ opererà sullo stato S_i e sul costo C_i. Se, invece, si verifica un rifiuto della soluzione prodotta alla i-esima iterazione, la iterazione successiva $i+1$ necessiterà dello stato S_{i-1} e del costo C_{i-1} calcolati dall'iterazione $i-1$.

Tutto ciò suggerisce la seguente tecnica di sfruttamento del parallelismo. Dati tre processori A, B, e C, possiamo incaricare A di eseguire l'iterazione i e, contemporaneamente, gli altri due di speculare sul suo esito; uno di essi, B, supponendo che l'iterazione i si concluda col rifiuto dello stato S_i, utilizzerà lo stato S_{i-1} ed il costo C_{i-1}, mentre il terzo processore C, ipotizzando l'accettazione dello stato S_i, lavorerà sullo stato S_i comunicatogli da A. Quando l'iterazione i sarà conclusa, in base al suo esito si saprà quale tra i due processori B e C avrà speculato correttamente e il suo stato diventerà lo stato corrente del sistema.

Con questa tecnica, in un tempo paragonabile a quello richiesto per calcolare una singola iterazione con un solo processore, tre processori ne calcoleranno due. In Figura 4.5 è mostrato lo schema logico di questa organizzazione.

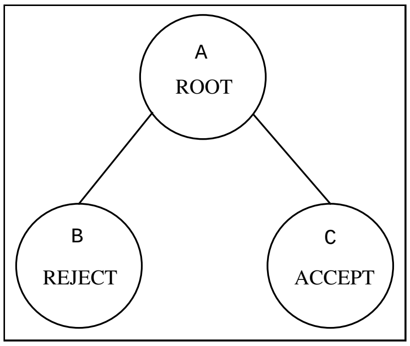


Figura 4.5: Albero binario con tre processori

```
Program master(initial_solution : solution):solution;
begin
current_solution = initial_solution;
current_cost = evaluate(current_solution);
T = T_initial;
while (T > T_final)
begin

        for i = 0 to (iterations(T)/p)
        begin

                send(root, current_solution);
                send(root, current_cost, T);
                receive(any_leaf, current_solution, current_cost);

        end;
        T = new_temp(T);

end;
return(current_solution);
end
```

Figura 4.6: Algoritmo eseguito dal nodo *master*, dal lavoro di Perego.

Questo schema implementativo sfrutta efficientemente la tecnica del *calcolo speculativo*: una computazione viene fatta procedere ancor prima di saper se essa sarà necessaria; in caso affermativo parte del lavoro sarà già stato fatto quando esso sarà richiesto per l'avanzamento dell'elaborazione, in caso contrario il lavoro fatto risulterà inutile e dovrà essere scartato.

È abbastanza semplice estendere lo schema implementativo descritto per tre processori ad un insieme di $P = 2^p$ nodi di elaborazione $(P-1)$ dei quali logicamente interconnessi a formare un albero binario completo di p livelli, e il processore rimanente, che chiameremo *master*, con funzioni di controllo dell'algoritmo SA. In questo caso $p = log_2 P$ iterazioni saranno eseguite contemporaneamente con uno *speedup* ideale di $log_2 P$ rispetto alla versione sequenziale.

L'implementazione di questa versione parallela dell'algoritmo di SA è descritta dallo pseudocodice riportato nelle Figure 4.6 e 4.7. L'algoritmo riportato in Figura 4.6 è eseguito dal processore *master*.

Quest'ultimo gestisce l'abbassamento della temperatura e il ciclo itera-

```
Program Tree_Node();
begin
mypid = whoami();
*[receive(parent, current_solution) - - >

        if Not_a_Leaf(mypid)then send(Reject_son, current_solution);
        new_solution =move(current_solution);
        if Not_a_Leaf(mypid) then send(Accept_son, new_solution);
        new_cost = evaluate(new_solution);

[] receive(parent, (current_cost, T)) - - >

        ΔCost = new_cost - current_cost;
        if((ΔCost ≤ 0) ⋁((e^{ΔCost/T}) >random()))then
        begin

                current_solution = new_solution;
                current_cost = new_cost;
                right_son = Accept_son;

        end
        else right_son = Reject_son;
        if Not_a_Leaf(mypid)then send(right_son, (current_cost, T));
        else send(master, (current_solution, current_cost);

]^a ;
end
```

[a] *[...[] ...] indica un comando non deterministico ripetitivo in stile CSP

Figura 4.7: Algoritmo eseguito dai nodi dell'albero binario di processori, dal lavoro di Perego.

tivo viene eseguito $iterations(T)/p$ volte per ogni temperatura T. Ad ogni iterazione i, il $master$ invia al processore radice dell'albero binario la soluzione corrente S_i, il costo C_i e il valore della temperatura attuale T; quindi si pone in attesa non deterministica del risultato S_{i+p}, C_{i+p} da una delle foglie dell'albero binario

$$receive(any\_leaf, current\_solution, current\_cost).$$

I processori posti sui nodi dell'albero binario eseguono l'algoritmo riportato in Figura 4.7.

Tale algoritmo può essere suddiviso in due fasi logiche distinte eseguite alternativamente. Le due fasi corrispondono alla fase di propagazione delle soluzioni, che interessa tutti i processori, e alla fase decisionale, che interessa invece solo i processori posti sul cammino seguito dal flusso decisionale che collega la radice dell'albero ad una delle foglie.

La fase di **propagazione delle soluzioni** prevede che alla ricezione della soluzione S_i dal nodo padre, ogni processore posto su un nodo interno dell'albero esegua i seguenti passi:

- propaga immediatamente la soluzione al figlio $Reject\_Son$ che specula sul rigetto della soluzione provata dal padre;
- genera la nuova soluzione S_{i+1} ;
- la invia al figlio $Accept\_Son$ che specula sull'accettazione di S_{i+1};
- valuta il costo C_{i+1} di S_{i+1}.

Chiaramente i processori posti sulle foglie dell'albero non propagano la soluzione ricevuta ma eseguono solamente il secondo ed il quarto passo.

La fase di **propagazione delle decisioni** prevede che alla ricezione della coppia $(current\_cost, T)$ dal nodo padre, ogni processore posto su un nodo interno dell'albero attraversato dal flusso decisionale decida l'esito del proprio tentativo: nel caso la nuova soluzione locale non venga accettata propaga la coppia $(current\_cost, T)$ al figlio $Reject\_son$; in caso di accettazione, la nuova coppia (new_cost, T) viene inviata al figlio $Accept\_Son$. Il processore foglia dell'albero raggiunto dal flusso decisionale, dopo aver deciso sul proprio tentativo, comunica la coppia (S_{i+p}, C_{i+p}) al processore *master* permettondogli così di aggiornare lo stato del sistema ed eventualmente di iniziare il blocco di iterazioni successive.

L'autore ha migliorato ulteriormente l'algoritmo sopra descritto attraverso due semplici considerazioni:

- lo *speedup* di un algoritmo parallelo di SA che adotti la tecnica del calcolo speculativo è limitato superiormente dalla profondità dell'albero o, in altre parole, dal numero di iterazioni che vengono eseguite concorrentemente;

- il fattore di accettazione varia ampiamente in funzione della temperatura del sistema. Infatti alle alte temperature la probabilità di accettare le nuove configurazioni è molto elevata mentre decresce fino ad avvicinarsi allo zero man mano che ci si avvicini alla temperatura finale.

L'autore ha cercato di massimizzare la lunghezza del cammino percorso dal flusso di decisione (e quindi il grado di parallelismo) rinunciando al bilanciamento dell'albero la cui struttura deve essere modificata dinamicamente in funzione della temperatura corrente del sistema. Ad esempio, alle alte temperature converrà sfruttare tutti i nodi disponibili per speculare sull'accettazione delle configurazioni generate, mentre, durante gli ultimi passi dell'algoritmo parallelo, sarà conveniente concentrare tutti i processori sul cammino corrispondente ad una sequenza decisionale di soli rigetti delle mosse tentate.

È stata adottata una semplice euristica tesa a costruire dinamicamente l'albero logico dei processori che massimizza la lunghezza del cammino seguito dal flusso di decisione. Il fatto di costruire dinamicamente le connessioni logiche tra i processori anzichè utilizzare un insieme di schemi predefiniti aumenta sensibilmente la flessibilità e adattabilità dell'approccio che diventa indipendente dal numero di processori usati e dal procedimento di raffredamento del sistema scelto.

4.3 Conclusioni

In questo capitolo sono state esaminate alcune soluzioni al problema della parallelizzazione dell'algoritmo di SA.

L'algoritmo di SA, per quanto riguarda la sequenza di accettazione e rifiuto delle soluzioni, è implicitamente sequenziale. Per sopperire a questa caratteristica che ne limita la parallelizzabilità, alcune soluzioni esaminate [3], [11] rilasciano il vincolo della serialità della sequenza di accettazione o rifiuto delle soluzioni di un algoritmo sequenziale permettendo a più processori la contemporanea modifica della configurazione del sistema. Questa tecnica porta ad un incremento dello *speedup* degli algoritmi, ma introduce del disturbo nella fase decisionale che generalmente viene fatta localmente a partire da un'energia del sistema che può essere inconsistente a causa dell'accettazione di modifiche allo stato apportate da altri nodi di elaborazione.

Alcune soluzioni proposte nel capitolo fanno comunicare periodicamente i vari algoritmi di SA in esecuzione parallela, in modo da mantenere l'ammissibilità delle soluzioni. Le comunicazioni possono diventare in questo caso dei colli di bottiglia.

Le implementazioni [46], [52] provano a speculare sul risultato del processo di accettazione o rifiuto dell'algoritmo di SA, calcolando entrambe le possibilità in parallelo, in modo da effettuare del lavoro utile nel seguito del processo di elaborazione. Queste implementazioni mantengono il vincolo della stretta sequenzialità e sono indipendenti dal problema trattato ma hanno lo svantaggio che lo *speedup* è limitato superiormente dalla profondità dell'albero.

Capitolo 5

Studio degli AG per il TSP

In questo Capitolo presentiamo i risultati dello studio effettuato sull'applicazione degli AG al problema del commesso viaggiatore [5], che in seguito indicheremo con l'abbreviazione TSP. Per una descrizione del TSP e degli operatori genetici per questo tipo di problema, si vedano rispettivamente le Appendici C e B.

Gli esperimenti sono stati eseguiti sull'elaboratore parallelo nCUBE2; per maggiori dettagli sull'architettura dell'elaboratore si veda l'Appendice A.

La prima fase dello studio ha riguardato l'implementazione e la prova di AG sequenziali, allo scopo di effettuare una preliminare valutazione sui diversi criteri di rimpiazzamento degli individui all'interno della popolazione, sui diversi valori del parametro di *crossover* e sul tipo di operatore di *crossover*. Sono stati implementati due operatori di *crossover* e tre tipi di rimpiazzamento dei genitori con i figli, ed è stata studiata l'influenza di questi operatori genetici sulla bontà della soluzione.

I test effettuati hanno mostrato il comportamento degli algoritmi, evidenziando quale scelta degli operatori genetici di rimpiazzamento e di *crossover* conduce ad una migliore convergenza dell'algoritmo. L'operatore di *crossover* che ha mostrato un migliore comportamento è stato utilizzato negli AG paralleli. Sono stati implementati, secondo il modello di programmazione SPMD (*Single Program Multiple Data*) [4], AG paralleli a *grana fine* e a *grana grossa*, e ne sono state misurate le prestazioni al variare di alcuni parametri. Il modello SPMD prevede che tutti i nodi eseguano lo stesso programma sequenziale su differenti dati del problema.

L'algoritmo genetico a *grana fine* adotta uno schema di *mapping* che consente di far variare in modo indipendente la dimensione della popolazione e il numero di nodi sui quali l'algoritmo viene eseguito.

Infine è stato effettuato un confronto tra gli algoritmi a *grana fine* e a *grana grossa* con l'algoritmo di SA presentato in [46], opportunamente modificato per trattare il problema TSP.

5.1 AG sequenziali

Il modello di generazione da noi adottato nell'implementazione degli AG sequenziali è il *modello di generazione discreto* [20] (vedi Paragrafo 1.3) in cui la popolazione di genitori viene tenuta distinta dalla popolazione dei figli. Quando i figli sono stati tutti generati, entrano a far parte della popolazione di genitori seguendo il criterio di rimpiazzamento adottato.

Gli AG sono stati implementati usando la rappresentazione del TSP detta a *percorso* (vedi Appendice B): i percorsi sono rappresentati da una sequenza ordinata di città, e le città contigue sono quelle unite da un arco. Gli operatori genetici operano su un insieme ordinato di numeri, il cui campo di variabilità va da 0 al numero delle città del particolare problema rappresentato. La rappresentazione a *percorso*, pur avendo il vantaggio della chiarezza, richiede l'uso di operatori genetici modificati. Infatti, applicando il normale operatore di *crossover*, si corre il rischio di creare percorsi non legali (città ripetute e percorsi interrotti).

Si sono implementati due diversi operatori di *crossover* e tre modalità di rimpiazzamento dei padri con i figli. Sia negli AG sequenziali che in quelli paralleli, descritti nel Paragrafo 5.2, il criterio adottato per la selezione degli individui sui quali applicare l'operatore di *crossover* per produrre figli è casuale. Nei Paragrafi seguenti si descrivono gli operatori genetici e le modalità di rimpiazzamento adottate.

5.1.1 Operatore di mutazione

L'operatore di mutazione implementato per la rappresentazione a *percorso* è un operatore già usato in [20]. Esso non fa altro che scegliere due distinte città della stringa che definisce il *percorso* corrente, e scambiarle di posto. Per esempio, avendo un problema TSP a 6 città, definito dalla soluzione corrente

$$A = (123456)$$

ed applicando l'operatore di mutazione alle città 3 e 4, si ottiene

$$A' = (124356)$$

che, come si vede nella Figura 5.1, è ancora una soluzione ammissibile del problema.

5.1.2 Operatore di crossover

L'operatore di *crossover* per la rappresentazione a *percorso* deve essere tale da non generare percorsi illegali. Infatti, applicando semplicemente la definizione originaria di Holland, si ottengono percorsi con città isolate, quindi non legali. Per esempio, sempre trattando un TSP a 6 città, supponiamo di avere i due genitori (si veda la Figura 5.2):

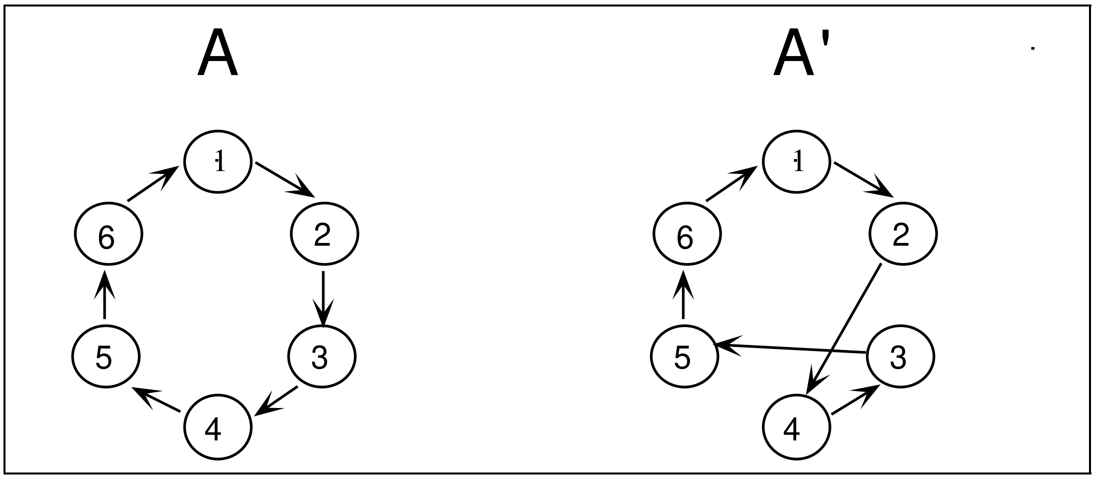


Figura 5.1: Esempio di funzionamento dell'operatore di mutazione.

$$A = (432156)$$

e

$$B = (123465)$$

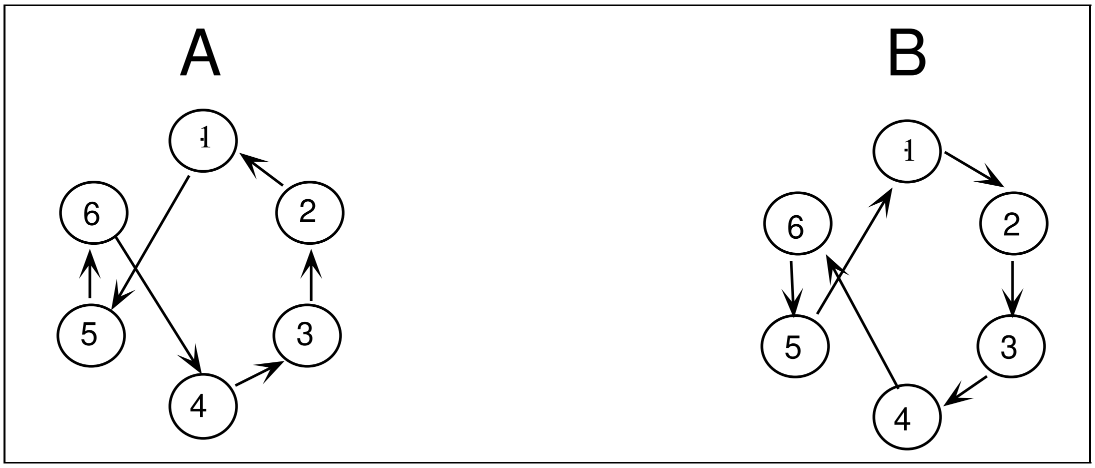


Figura 5.2: Esempio di applicazione dell'operatore di *crossover* al problema TSP con 6 città.

e scegliamo il punto di *crossover* a metà, cioè dopo la terza città. I figli generati sono quindi:

$$C = (432|465)$$

e

$$D = (123|156).$$

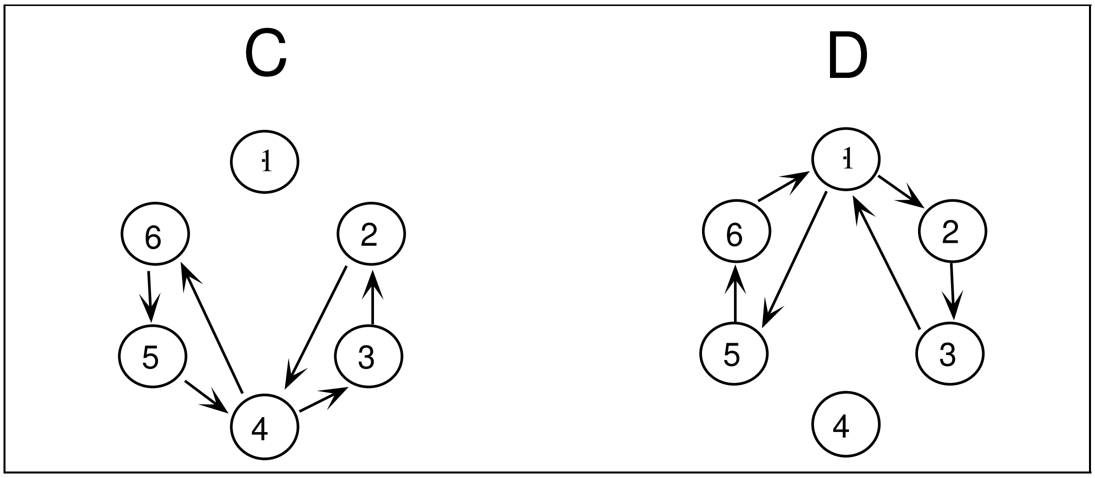


Figura 5.3: Esempio di creazione di percorsi illegali tramite l'applicazione dell'operatore di *crossover*.

Dalla Figura 5.3 risulta chiaro come entrambi i figli C e D rappresentino percorsi illegali. In D la città 4 non ha archi entranti o uscenti, e la città 1 ha due archi di troppo. In C si nota come la città 1 risulti isolata dalle altre, e come la città 4 abbia un arco di troppo sia in entrata che in uscita.

Nel seguito vengono esposte le soluzioni adottate per consentire la creazione di percorsi legali mediante l'applicazione dell'operatore di *crossover*.

Operatore di crossover ad un punto

L'operatore di *crossover ad un punto* [7], opera spezzando il codice genetico dei genitori in uno specifico punto, e ricombinando nei figli le informazioni contenute nelle due metà. Il crossover implementato evita la creazione di percorsi illegali controllando, per ogni città che deve essere inserita nel *percorso* del figlio, se questa non sia già stata inserita precedentemente dall'altro genitore. Se risulta già inserita, l'operatore di *crossover* passa alla prossima città del padre. La legalità dei percorsi dei genitori, rende sempre possibile trovare nel padre il numero di città necessarie a completare correttamente il *percorso* del figlio. Rifacendoci all'esempio precedente, il figlio C sarebbe stato creato dall'operatore di *crossover ad un punto* nel seguente modo. Dapprima il figlio C avrebbe ereditato il codice genetico del padre A fino al punto di *crossover*, cioè fino alla città 2 compresa

$$C = (432 * **);$$

quindi l'operatore avrebbe considerato l'altro genitore B dal punto di crossover in poi (cioè le città dalla 4 in poi). Ma, come si vede esaminando la stringa che definisce C, la città 4 risulta già presente nel figlio. Quindi l'operatore di *crossover* avrebbe continuato con la successiva città del padre B, che è 6. Questa non è presente nel figlio C, e può quindi in esso essere inserita ottenendo:

$$C = (4326 * *).$$

Successivamente C viene completato con le due città 5 e 1 del padre B.

$$C = (432651).$$

Come mostrato in Figura 5.4, il *percorso* ottenuto è valido.

Operatore di crossover a due punti

L'operatore di *crossover a due punti* [16] opera spezzando il codice genetico in tre parti, e ricombinandole nei figli in modo da formare percorsi legali.

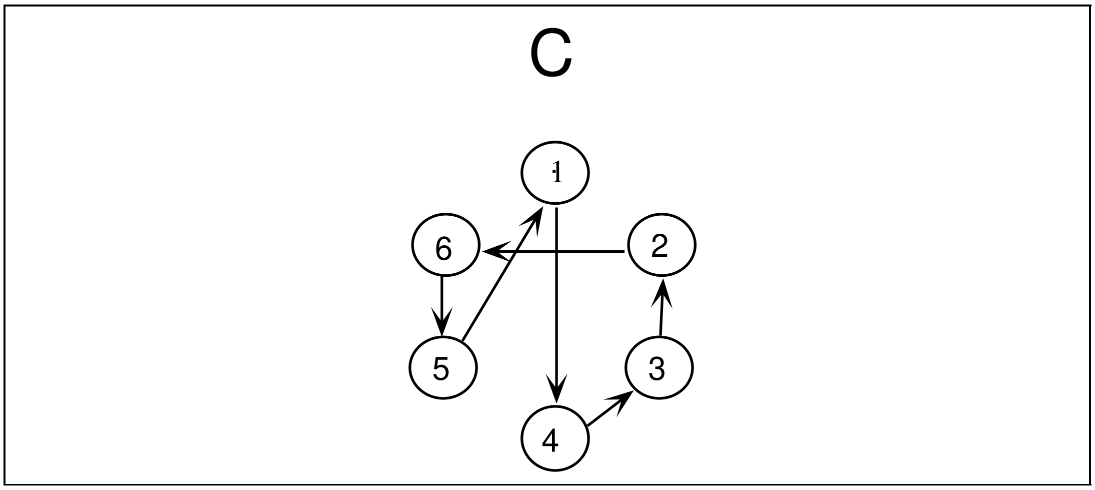


Figura 5.4: Esempio di corretta applicazione dell'operatore di *crossover* ad un punto.

L'operatore funziona in modo simile al *crossover ad un punto*, controllando per ogni città che deve essere inserita nella stringa del figlio, se questa non sia già stata inserita precedentemente. Supponiamo di avere un problema TSP a 6 città, con i genitori

$$A = (43|21|56)$$

$$B = (62|34|15)$$

rappresentati in Figura 5.5 e scegliamo i punti di *crossover* dopo la seconda e la quarta città.

Il genitore A viene quindi suddiviso nelle tre stringhe (43), (21) e (56). Queste costituiscono la *parte iniziale*, *centrale* e *finale* del codice genetico di A. Dapprima, copiando la *parte centrale* del codice genetico del genitore A nella corrispondente posizione del figlio, si crea la *parte centrale* del figlio C

$$C = (**21**).$$

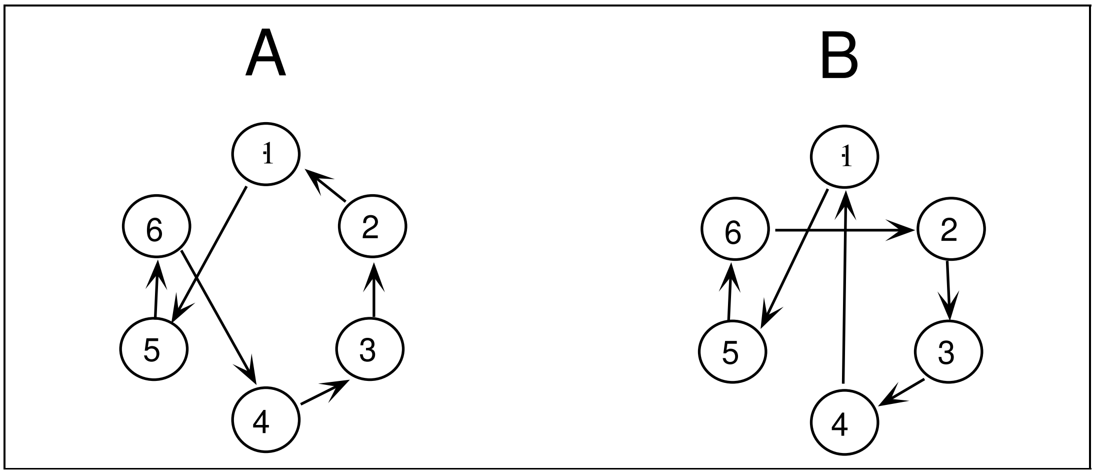


Figura 5.5: Esempio di applicazione dell'operatore di *crossover a due punti* ad un problema TSP con 6 città.

Quindi si considera la *parte iniziale* di B, cioè le prime due città del genitore B, che sono (62). La città 6 può essere inserita nella prima posizione del figlio C, mentre la città 2 viene scartata perchè risulta già presente nella stringa. Il figlio risulta così composto:

$$C = (6 * 21 * *).$$

Come si vede la *parte iniziale* del figlio C deve ancora essere completata. Questa operazione sarà fatta nel seguito. Per adesso ci occupiamo di come riempire la *parte finale* del figlio C. A questo scopo consideriamo la *parte finale* del genitore B, cioè le sue ultime due città (15). Mentre la città 1 non può essere inserita in C perchè in essa già presente, la città 5 viene posta al termine della stringa del figlio, che risulta essere quindi

$$C = (6 * 21 * 5).$$

Come si vede, anche la *parte finale* di C deve essere completata. È ora necessario completare le parti *iniziale* e *finale* della stringa del figlio C con le città mancanti. Tali città vengono prelevate dalla *parte centrale* della stringa del genitore B; questa scelta garantisce la reperibilità di tutte le città necessarie a completare correttamente il *percorso*. Infatti, nell'esempio, la *parte centrale* di B è costituita dalle città (34) che, inserite nel figlio C, portano alla formazione di un *percorso* legale, come mostrato in Figura 5.6

$$C = (632145).$$

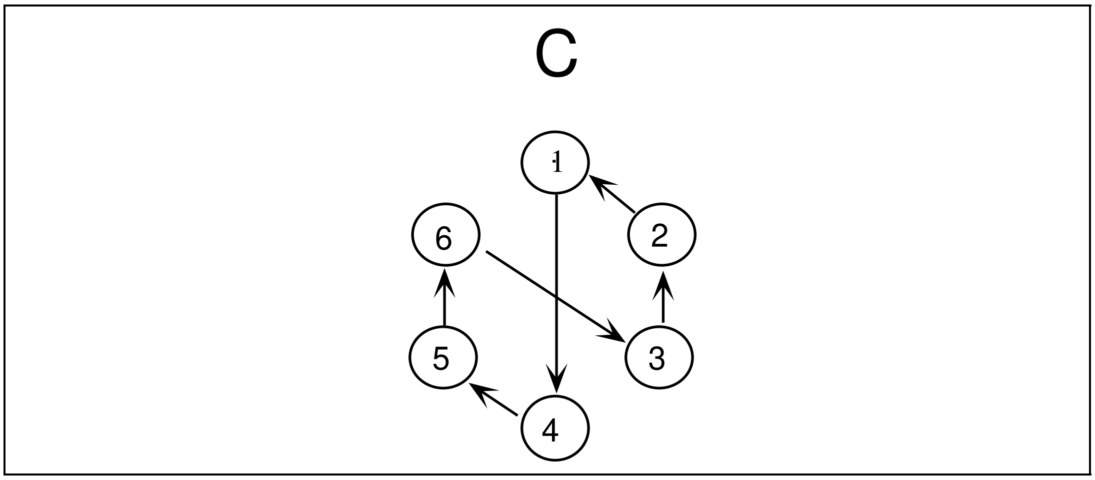


Figura 5.6: Esempio di corretta applicazione dell'operatore di *crossover* a due punti.

Modalità di rimpiazzamento

I criteri di rimpiazzamento stabiliscono le modalità con le quali i figli che vengono generati mediante l'applicazione dell'operatore di *crossover* entrano nella popolazione di soluzioni. Le soluzioni implementate sono tre, e verranno chiamate nel seguito R1, R2 e R3.

Le tre modalità funzionano nel seguente modo:

- modalità di rimpiazzamento R1: sostituisce gli individui della popolazione che hanno alti valori di *fitness* [1]. Dal momento che la popolazio-

[1] Ricordiamo qui che il problema trattato richiede di trovare il minimo percorso che unisce tra loro tutte le città. Un alto valore di *fitness* indica un percorso più lungo di altri, e quindi la soluzione corrispondente può essere scartata.

ne di individui viene mantenuta ordinata in base al valore di *fitness*, R1 non fa altro che sostituire nella popolazione tutti gli individui situati nelle posizioni contraddistinte da un'alta *fitness* con i figli appena creati. In questo modo si possono introdurre nella popolazione anche dei figli che hanno una *fitness* più alta dei padri che stanno rimpiazzando. D'altronde proprio questo fatto introduce una maggiore variabilità nel codice genetico complessivo della popolazione, e ciò può risultare utile per esplorare un maggior numero di *regioni* dello spazio delle soluzioni (vedi Paragrafo 1.4);

- modalità di rimpiazzamento R2: ordina dapprima i figli in base al valore di *fitness*, per poi confrontare il figlio con minor valore di *fitness* con l'elemento della popolazione che ha il più alto valore di *fitness*. Se il figlio risulta avere una *fitness* migliore (più bassa), allora rimpiazza l'elemento della popolazione, e va avanti considerando il prossimo figlio e il successivo elemento della popolazione. In tal modo si rimpiazzano gli elementi peggiori della popolazione con i figli appena creati solamente se i figli hanno migliori valori di *fitness*. Si controlla così maggiormente l'andamento della popolazione di soluzioni lungo il susseguirsi delle generazioni, permettendo solo ai migliori di entrarne a far parte;

- modalità di rimpiazzamento R3: utilizza la *fitness media* della popolazione per decidere se un figlio deve rimpiazzare o meno un elemento della popolazione. Tra tutti i figli generati, vengono selezionati solo quelli che hanno un valore di *fitness* inferiore alla media (portano cioè soluzioni migliori della media della popolazione). Questi figli vanno a rimpiazzare solamente gli elementi della popolazione che hanno un valore di *fitness* superiore alla media.

5.2 AG paralleli

5.2.1 Modello a grana grossa

Il modello di generazione adottato nell'implementazione degli AG paralleli a *grana grossa* è il *modello di generazione discreto* (vedi Paragrafo 1.3), in cui i figli generati vengono inseriti nella popolazione successiva. Come risulta da [20], tale modello di generazione risulta essere il più adatto per gli AG a *grana grossa*. L'implementazione adotta il modello *stepping stone* [20] (vedi Paragrafo 2.1.3), con le sottopopolazioni connesse tramite una struttura ad anello. Le migrazioni avvengono in un solo senso e ad intervalli uguali per tutte le sottopopolazioni. Gli individui da trasferire alle altre sottopopolazioni sono scelti tra quelli che forniscono il miglior valore di *fitness*. Lo pseudocodice dell'algoritmo a *grana grossa* implementato è riportato in Figura 5.7.

La popolazione complessiva è stata fissata in N individui, da suddividere in base al numero di processori usati per l'esecuzione dell'algoritmo. Il numero di individui da migrare è una percentuale fissa della sottopopolazione. Le migrazioni, seguendo il criterio proposto in [10], avvengono periodicamente ad intervalli regolari per ciascuna sottopopolazione, dopo che si sono eseguite un certo numero di generazioni.

La modalità adottata per il rimpiazzamento degli elementi della sottopopolazione con gli individui migrati da altre sottopopolazioni è R1 allo scopo di immettere tutti i migranti all'interno della sottopopolazione. Essendo il numero di individui da migrare un parametro critico[2] negli AG a *grana*

[2]Si intende che il parametro di migrazione può influenzare la convergenza degli AG verso l'ottimo.

```
Program GRANA_GROSSA;
begin
whoami(mionodo,M);
/*Determina l'identificatore del nodo e la dimensione dell'ipercubo */
num_processori = 2^M;
num_individui = TOTALE_INDIVIDUI div num_processori;
num_figli = parametro_crossover * num_individui;
num_mutazioni = parametro_mutazione * num_individui;
popolazione=GENERA(num_individui);
/* Inizializza la sottopopolazione di num_individui su ogni nodo*/
ORDINAMENTO(popolazione);
CALCOLA_FITNESS(popolazione);
for epoche = 1 to num_epoche do
begin

      for gen = 1 to num_gen do
      begin

            for i = 1 to (num_figli div 2) do
            begin

                  genitore1=SELEZIONA(popolazione);
                  genitore2=SELEZIONA(popolazione);
                  CROSSOVER(genitore1,genitore2,figlio1,figlio2);
                  /* Il crossover produce dai medesimi genitori due figli diversi */
                  AGGIORNA_POP_FIGLI(figlio1,figlio2,pop_figli);
                  /* i figli generati vengono messi nella popolazione dei figli*/


            end;
            for i = 1 to (num_mutazioni) do
            begin

                  figlio_da_mutare=SELEZIONA(pop_figli);
                  MUTAZIONE(figlio_da_mutare);

            end;
            CALCOLA_FITNESS(pop_figli);
            ORDINAMENTO(pop_figli);
            /*Ordina i figli in base al valore crescente di fitness*/
            RIMPIAZZAMENTO(popolazione,pop_figli);
            /*I figli rimpiazzano solo gli individui della popolazione con valore di fitness peggiore */

      end; /* fine ciclo di num_gen*/
      migranti=SELEZIONA_INDIVIDUI_MIGLIORI(popolazione);
      INVIA_AL_NODO(migranti);
      RICEVI_DAL_NODO(migranti);
      /*invia al nodo successivo e riceve dal nodo precedente*/
      RIMPIAZZA_MIGRANTI(popolazione,migranti);
      /*Gli individui ricevuti rimpiazzano gli individui peggiori della popolazione */

end; /* fine ciclo di num_epoche */
end
```

Figura 5.7: Pseudocodice dell'algoritmo genetico parallelo a *grana grossa*.

grossa, si sono rilevate le prestazioni dell'algoritmo al variare del parametro di migrazione.

5.2.2 Modello a grana fine

Il modello di generazione adottato nell'implementazione degli AG paralleli a *grana fine* è il *modello di generazione continuo* (vedi Paragrafo 1.3), in cui i figli generati vengono inseriti immediatamente nella popolazione corrente. Come risulta da [20], tale modello risulta essere il più adatto per gli AG a *grana fine*.

Nel modello a *grana fine* la popolazione è strutturata secondo una topologia logica che definisce le possibilità di interazione di un individuo con il resto della popolazione: ogni individuo s è posto su un vertice $v(s)$ della topologia logica T, e può accoppiarsi solamente con gli individui posti nell'*intorno* costituito dai vertici direttamente connessi a $v(s)$ in T. Nel nostro caso la popolazione di 2^N individui ha una struttura logica ad *ipercubo* N-dimensionale. Tale scelta, oltre ad essere direttamente correlata all'omomorfismo con la topologia della macchina fisica utilizzata per le sperimentazioni, permette di ovviare in maniera semplice ed efficiente alla principale carenza di buona parte delle implementazioni del modello a *grana fine* che vincolano il numero di individui della popolazione al numero di processori fisici disponibili. Sfruttando la definizione ricorsiva della topologia ad ipercubo si possono invece rendere indipendenti questi due parametri. Come si può vedere dall'esempio schematizzato in Figura 5.8, una popolazione di $2^3 = 8$ individui può essere allocata su un ipercubo di $2^2 = 4$ nodi semplicemente mascherando il primo (o l'ultimo) *bit* del codice **Grey** [44] usato per la numerazione dei vertici di un ipercubo. Il nodo fisico $X00$ conterrà gli individui 000 e 100 della topologia logica senza per questo violare in alcun modo le relazioni di vicinanza che determinano la strutturazione della popolazione. In altre parole, gli individui dell'intorno di un qualsiasi individuo s continueranno ad essere posti su vertici connessi a $v(s)$ anche nella topologia fisica.

Tale schema di *mapping* può essere chiaramente generalizzato: per determinare l'allocazione di una popolazione di 2^N individui su un ipercubo fisico di 2^M nodi con N qualsiasi e $M < N$ è sufficiente mascherare i primi (gli ultimi) $N - M$ *bit* della codifica binaria di ogni individuo.

Sfruttando questa proprietà della topologia ad ipercubo, è semplice disegnare un'implementazione parallela di un algoritmo genetico a *grana fine*, quale quella descritta dallo pseudocodice di Figura 5.9, in cui il numero di individui della popolazione ($TOTALE\_INDIVIDUI$) ed il numero di processori utilizzati ($num\_processori$) siano parametri indipendenti.

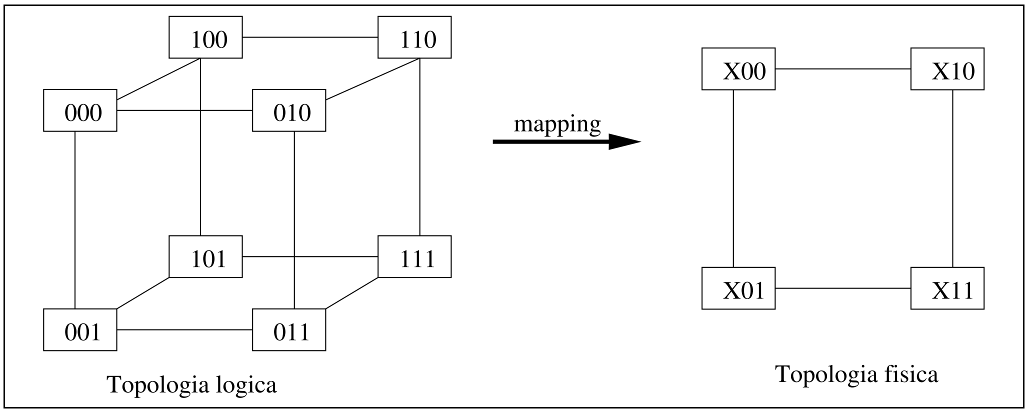


Figura 5.8: Esempio di applicazione dello schema di *mapping* per l'algoritmo genetico a *grana fine*.

5.3 Risultati sperimentali

Ogni AG, sequenziale e parallelo, è stato applicato alla risoluzione di due problemi TSP di dimensioni pari a 48 città e 105 città. I problemi utilizzati

```
Program Grana_Fine;
begin
whoami(mionodo,M);
/*Determina l'identificatore del nodo e la dimensione dell'ipercubo allocato */
TOTALE_INDIVIDUI = 2^N;
num_processori = 2^M;
num_individui = TOTALE_INDIVIDUI div num_processori;
popolazione=GENERA();
MAPPING(popolazione,num_individui,mionodo);
/*Determina su ogni nodo quali sono gli individui vicini allocati */
while condizione_di_terminazione do
begin

        INVIA_A_NODI_VICINI();
        RICEVI_DA_NODI_VICINI()();
        /* Scambio degli individui con i nodi fisicamente connessi */
        for cella = 1 to num_individui do
        begin

                contenuto_cella=SELEZIONA(popolazione);
                /* Seleziona un individuo tra quelli assegnati a mionodo */
                individuo_vicino=SELCASUALE(popolazione);
                /* Seleziona un individuo vicino tra gli individui che sono nell'intorno */
                CROSSOVER(individuo_vicino,contenuto_cella,figlio1,figlio2);
                /* Genera due figli applicando il crossover su individuo_vicino e
                contenuto_cella */
                figlio_da_mutare =SELFIGLIO(figlio1,figlio2);
                /*Seleziona uno dei due figli per applicare la mutazione*/
                MUTAZIONE(figlio_da_mutare);
                COMPUTA_FITNESS(figlio1,figlio2);
                /* Calcola i valori di fitness dei figli generati */
                contenuto_cella= RIMPIAZZAMENTO(contenuto_cella,figlio1,figlio2);
                /*Seleziona l'individuo che deve occupare la posizione cella: viene selezionato
                l'individuo con valore di fitness più basso*/

        end;


end;
end
```

Figura 5.9: Algoritmo genetico a *grana fine* espresso in pseudocodice.

per le prove sono noti come **GR48**, di soluzione ottima pari a 5046, e di **LIN105**, di soluzione ottima pari a 14379[3]. Per ogni AG sono state condotte 32 prove, utilizzando ogni volta popolazioni iniziali casuali e diverse. Si sono ottenuti così risultati sufficientemente generali da poter essere assunti come comportamento medio. Delle prove effettuate si sono presi in considerazione i seguenti risultati:

- la media delle soluzioni trovate dagli AG: $MED = \frac{\sum_{i=1}^{32} F_{E_i}}{32}$, dove con F_{E_i} si indica la *fitness* migliore dell'esecuzione E_i dell'algoritmo;
- la migliore delle soluzioni trovate: $MIG = min\{F_{E_i}, i = 1....32\}$;
- la peggiore delle soluzioni trovate: $PEG = max\{F_{E_i}, i = 1....32\}$.

5.3.1 Risultati degli AG sequenziali

Le prove sugli AG sequenziali sono state effettuate mantenendo invariati i seguenti valori:

- popolazione = 640 individui;
- parametro di Mutazione = 0.2, che corrisponde ad una applicazione dell'operatore di mutazione sul 20% della popolazione;
- generazioni: 2000 per il TSP a 48 città e 3000 per il TSP a 105 città; ciò per consentire una migliore risoluzione del problema a 105 città.

Nella prima serie di esperimenti è stato studiato il comportamento degli AG al variare del tipo di crossover (ad un punto e a due punti) e della modalità di rimpiazzamento (R1, R2, R3), fissato il valore del parametro di *crossover*.

[3] Entrambi i problemi sono disponibili all'indirizzo: ftp://elib.zib-berlin.de/pub/mp-testdata/tsp/tsplib.html

Nella seconda serie di esperimenti abbiamo studiato l'influenza del valore del parametro di *crossover* (che determina il numero di figli prodotti in ogni generazione) al variare del tipo di rimpiazzamento, fissato il tipo di crossover.

Nella terza serie di esperimenti è stata valutata l'influenza delle modalità di rimpiazzamento, fissato sia il tipo che il valore del parametro di crossover.

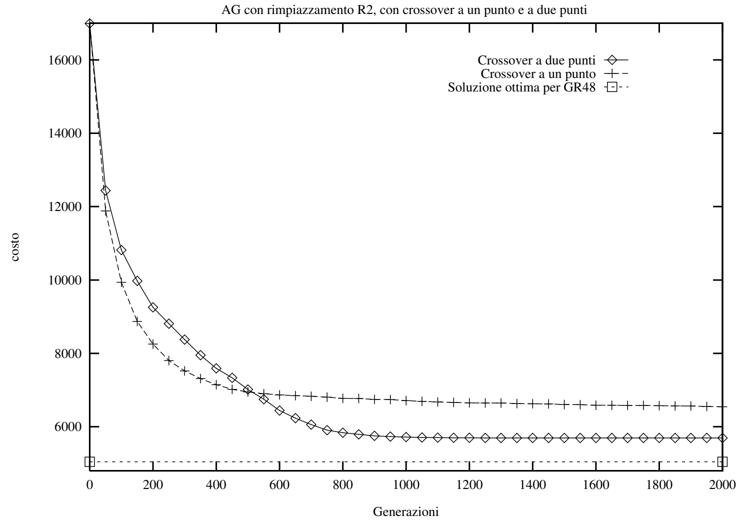
Nei grafici presentati nel seguito sono riportati i valori medi delle prove effettuate elaborando il problema TSP a 48 città all'aumentare del numero di generazioni. Le Tabelle riportano i valori medio (MED), migliore (MIG) e peggiore (PEG) delle prove effettuate sui problemi TSP a 105 e 48 città.

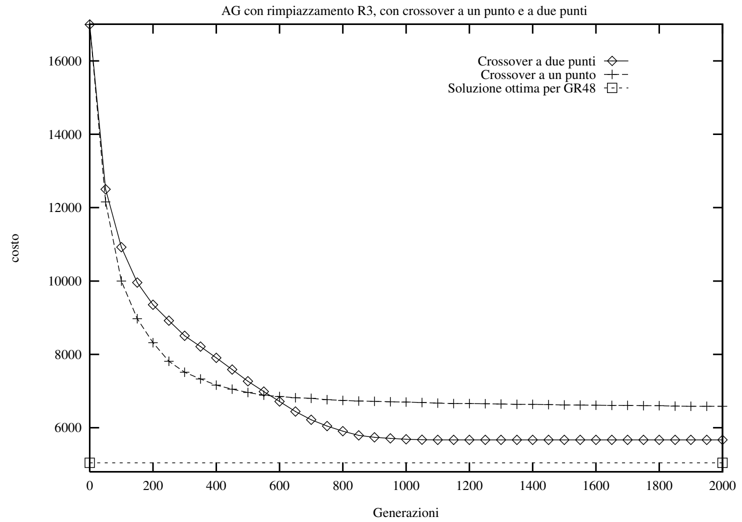
Tipo di crossover

I tipi di crossover adottati in questa serie di esperimenti sono due:

- crossover ad un punto;
- crossover a due punti.

In Figura 5.10 sono riportati i risultati medi delle prove effettuate. Il valore del parametro di *crossover* impiegato nelle prove è C=0.4, che corrisponde all'applicazione dell'operatore sul 40% della popolazione. Si può osservare che, in generale, il *crossover* a due punti converge ad una soluzione migliore del *crossover* ad un punto. Quest'ultimo presenta un comportamento iniziale migliore, ma converge poi a valori di *fitness* più alta. In particolare si può osservare che usando le modalità di rimpiazzamento R2 ed R3 il *crossover* a un punto fornisce soluzioni migliori del *crossover* a due punti fino a circa la generazione 500, mentre con la modalità di rimpiazzamento R1 il *crossover* ad un punto offre soluzioni migliori fino a circa la generazione 700 circa. Nelle Tabelle 5.1 e 5.2 sono riportati i risultati delle prove su 48 e su 105 città. Anche nel caso del problema a 105 città il *crossover* a due punti risulta fornire risultati migliori.





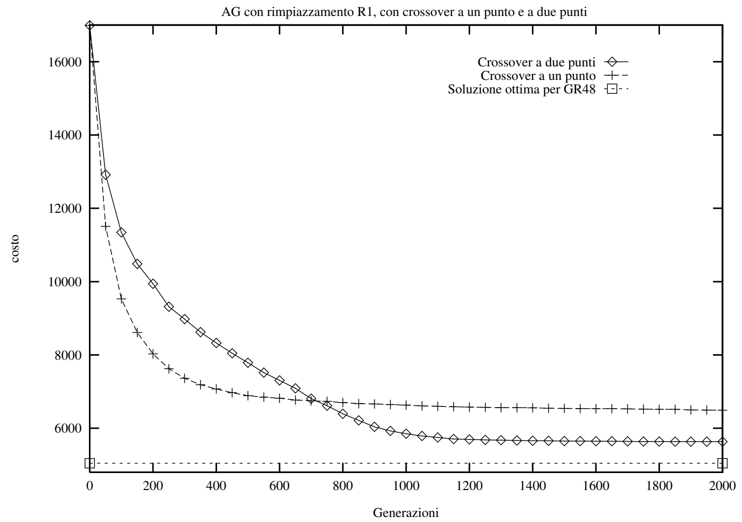


Figura 5.10: Valori di *fitness* ottenuti al variare della tecnica di rimpiazzamento (R1, R2, R3) e del tipo di *crossover*.

	crossover ad un punto			crossover a due punti		
	R1	*R2*	*R3*	*R1*	*R2*	*R3*
MED	6288	6547	6587	5632	5696	5669
MIG	5586	5717	5863	5315	5243	5178
PEG	8247	7623	7746	6079	6180	6140

Tabella 5.1: Valori di $fitness$ ottenuti con l'algoritmo genetico sequenziale al variare della tecnica di ripiazzamento (R1, R2, R3) e del tipo di *crossover* per il TSP a 48 città.

	crossover ad un punto			crossover a due punti		
	R1	*R2*	*R3*	*R1*	*R2*	*R3*
MED	25812	27045	27523	22857	22982	21864
MIG	21513	24222	24703	20587	18982	18942
PEG	29964	29874	32938	26176	23904	25115

Tabella 5.2: Valori di $fitness$ ottenuti con l'algoritmo genetico sequenziale al variare della tecnica di rimpiazzamento (R1, R2, R3) e del tipo di *crossover* per il TSP a 105 città.

Parametro di crossover

In Figura 5.11 si possono osservare i risultati delle prove condotte sulle tre modalità di rimpiazzamento R1, R2, R3 al variare del parametro di *crossover*. La variazione del valore del parametro di *crossover* influisce in maniera simile sulle modalità di rimpiazzamento R2 e R3: in entrambi i casi all'aumentare del parametro di *crossover* si ottiene una maggior velocità di convergenza.

Nel caso della modalità di rimpiazzamento R1 si ha un comportamento diverso: la maggior velocità di convergenza si ha per il parametro di $C = 0.6$. Infatti, data la modalità con la quale si effettuano i rimpiazzamenti, con $C = 0.8$ si inseriscono nella popolazione in maggior quantità figli che hanno un valore di $fitness$ peggiore dei padri. Dai dati riportati in Tabella 5.3, relativa al problema TSP a 48 città, si può osservare che per le modalità di rimpiazzamento R2 e R3 il miglior valor medio delle soluzioni si ottiene per $C = 0.4$, mentre per R1 si ha la soluzione media migliore per $C = 0.6$. Ciò è in accordo con quanto si osserva nei grafici di Figura 5.11: R1 ottiene soluzioni mediamente migliori per $C = 0.6$. In Tabella 5.4 sono riportati i dati relativi al problema a 105 città.

		Valore di C			
		0.2	0.4	0.6	0.8
R1	**MED**	6255	5632	5585	5870
	MIG	5510	5315	5135	5305
	PEG	7828	6079	6231	6693
R2	**MED**	5902	5696	5735	5743
	MIG	5405	5243	5323	5410
	PEG	7122	6180	6225	6243
R3	**MED**	6251	5669	5722	5773
	MIG	5441	5178	5281	5200
	PEG	7354	6140	6370	6594

Tabella 5.3: Valori di *fitness* ottenuti con l'algoritmo genetico sequenziale variando il parametro di *crossover* per il TSP a 48 città.

		Valore di C			
		0.2	0.4	0.6	0.8
R1	**MED**	34992	22857	22010	26955
	MIG	29997	20587	19450	23450
	PEG	39710	26176	25318	30865
R2	**MED**	32486	22982	20661	21154
	MIG	23423	18982	19176	18405
	PEG	39273	23904	22653	25815
R3	**MED**	28246	21864	22444	22735
	MIG	34399	18942	19600	18825
	PEG	42050	25115	25522	25778

Tabella 5.4: Valori di *fitness* ottenuti con l'algoritmo genetico sequenziale variando il parametro di *crossover* per il TSP a 105 città.

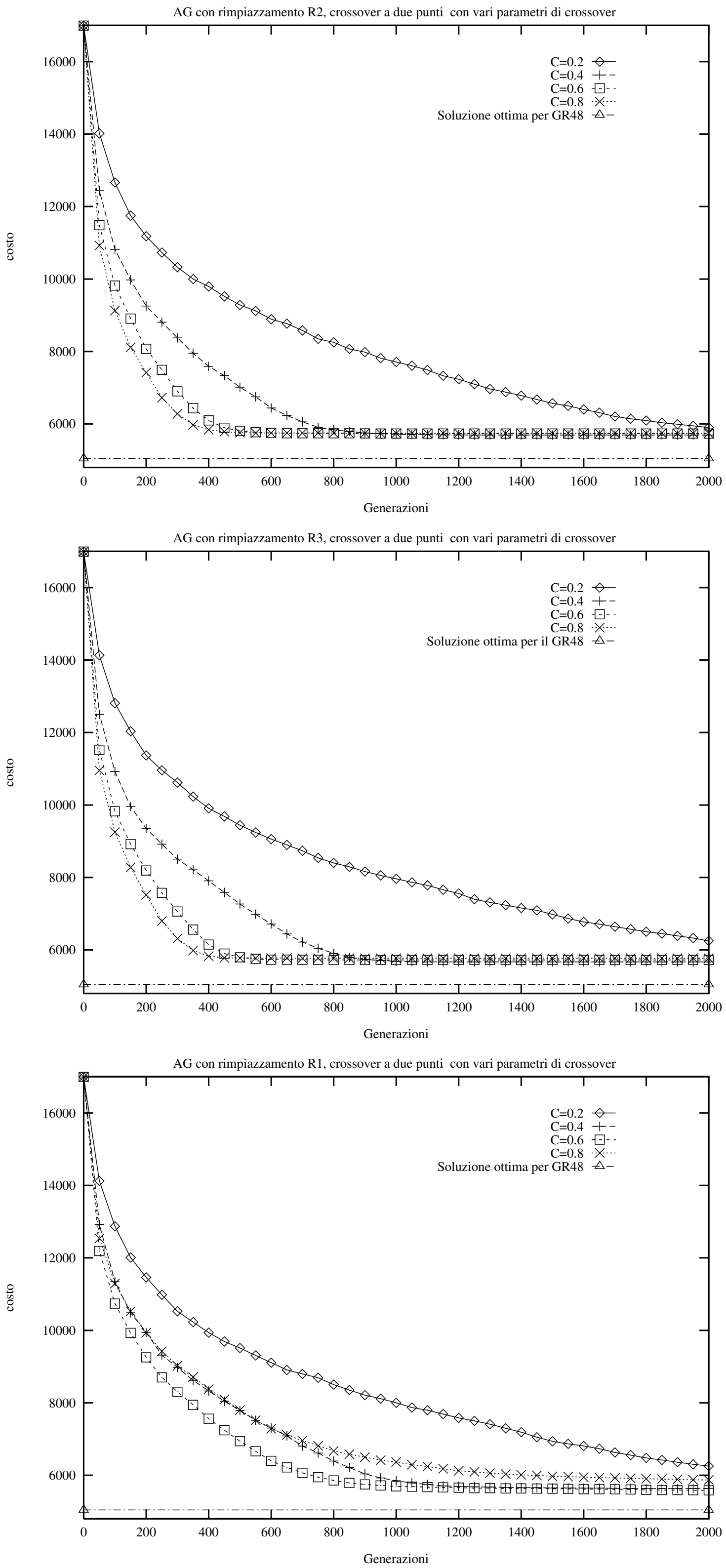


Figura 5.11: Valori di *fitness* ottenuti al variare della tecnica di rimpiazzamento adottata e del valore del parametro di *crossover*.

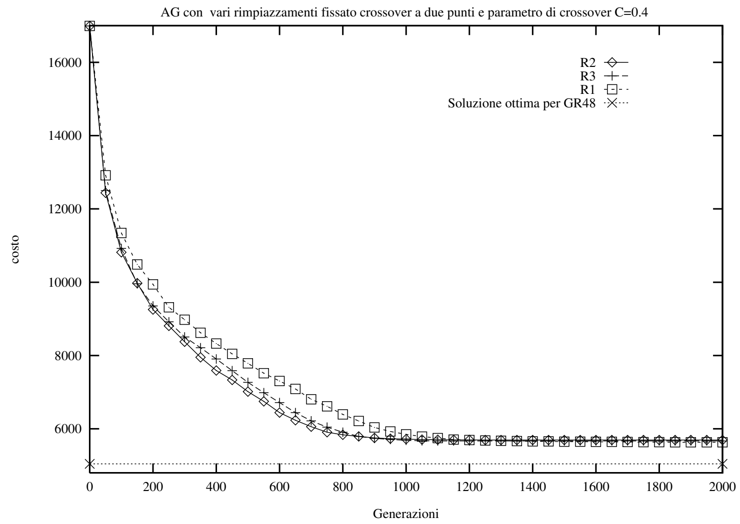


Figura 5.12: Valori di *fitness* ottenuti con l'algoritmo genetico sequenziale al variare della tecnica di rimpiazzamento (R1, R2, R3) con *crossover* a due punti, e parametro di *crossover* C=0.4.

Modalità di rimpiazzamento

Per il confronto delle modalità di rimpiazzamento si è scelto il valore $C = 0.4$ del paramentro di *crossover*, cioè il valore che ha dimostrato fornire i migliori risultati con i rimpiazzamenti R2 e R3 e che mostra un buon comportamento anche con R1. In Figura 5.12 si può osservare con maggiore chiarezza il confronto tra le diverse modalità di rimpiazzamento. Le modalità R2 e R3, come già notato in precedenza, convergono con maggiore velocità rispetto a R1. Inoltre R2 risulta avere una convergenza più veloce di R3.

Esaminando però i valori di *fitness* raggiunti dai tre rimpiazzamenti dopo 2000 generazioni (Tabella 5.3) si nota che R1 arriva ad un valore di *fitness* migliore sia di R2 che di R3, e che R3 consegue migliori valori di *fitness* di R2. Pertanto le modalità di rimpiazzamento che esibiscono una convergenza più lenta mostrano di avvicinarsi maggiormente all'ottimo.

5.3.2 Risultati degli AG paralleli

Modello a grana grossa

Dai *test* effettuati nei precedenti Paragrafi si deduce che il *crossover* a due punti porta ad una migliore approssimazione della soluzione ottima; inoltre la modalità di rimpiazzamento R2 ha mostrato una maggiore velocità di convergenza verso buoni valori di *fitness*. L'algoritmo parallelo adotta quindi il *crossover* a due punti e la modalità di rimpiazzamento R2. Nelle Tabelle 5.5 e 5.6 sono riportati i risultati dell'algoritmo genetico parallelo a *grana grossa* per i problemi a 48 e a 105 città. Osservando anche la Figura 5.13,

		Numero di nodi					
		N=4	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
M=0.1	**MED**	5780	5786	5933	6080	6383	6995
	MIG	5438	5315	5521	5633	5880	6625
	PEG	6250	6387	6516	6648	8177	8175
M=0.3	**MED**	5807	5877	5969	6039	6383	6623
	MIG	5194	5258	5467	5470	5727	6198
	PEG	6288	6644	7030	6540	8250	7915
M=0.5	**MED**	5900	5866	5870	6067	6329	6617
	MIG	5419	5475	5483	5372	6017	6108
	PEG	6335	6550	7029	6540	8250	7615

Tabella 5.5: Valori di *fitness* ottenuti con l'algoritmo genetico a *grana grossa* al variare del parametro di migrazione per il TSP a 48 città.

relativa al problema TSP a 48 città, si può notare che in generale si ha un peggioramento delle soluzioni media (MED), migliore (MIG) e peggiore (PEG) all'aumentare del numero di nodi. Ciò può essere spiegato osservando che la popolazione complessiva di 640 individui viene suddivisa in base al numero di nodi: su 4 nodi si hanno sottopopolazioni di 160 individui, mentre su 64 nodi la sottopopolazione è di soli 10 individui. Quindi al diminuire della sottopopolazione, si rileva un peggioramento della ricerca: sottopopolazioni con pochi individui non esplorano sufficientemente bene lo spazio delle soluzioni.

L'influenza del numero di individui da migrare è chiaramente osservabile notando come la soluzione media cambi al variare del parametro di migrazione M. Sul problema TSP a 48 città si nota che per un numero di nodi pari a N=4, 8, 16 il valore di migrazione che fornisce migliori soluzioni medie è M=0.1. Per un maggior numero di nodi invece il valore di M che fornisce migliori soluzioni medie è diverso: infatti per N=32 si ha M=0.3, e per N=64 si ha M=0.5. Quindi, al diminuire del numero di individui allocati su ogni nodo, un alto valore del parametro di migrazione riesce a migliorare le soluzioni trovate, rimescolando maggiormente il codice genetico delle sottopopolazioni.

Osservando i risultati riportati in Tabella 5.6, relativi al problema TSP a 105 città, si nota che la tendenza viene confermata: per N=4, 8 e 16 risulta M=0.3, mentre per N=32 e 64 risulta M=0.5.

		Numero di nodi					
		N=4	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
M=0.1	**MED**	23790	24538	27281	29364	32632	37872
	MIG	21370	21829	24532	26517	28966	36021
	PEG	26706	28381	37436	37008	37779	44048
M=0.3	**MED**	23560	24526	27063	29422	32542	35342
	MIG	20219	21120	23510	25783	30927	33015
	PEG	25899	29348	36795	39330	39131	41508
M=0.5	**MED**	23790	24670	26752	29137	32208	35067
	MIG	21272	21411	24088	23373	30617	31603
	PEG	26706	30875	36080	37779	36649	43494

Tabella 5.6: Valori di *fitness* ottenuti con l'algoritmo genetico a *grana grossa* al variare del parametro di migrazione per il TSP a 105 città.

In Tabella 5.7 sono riportati i tempi medi di esecuzione e gli indici di

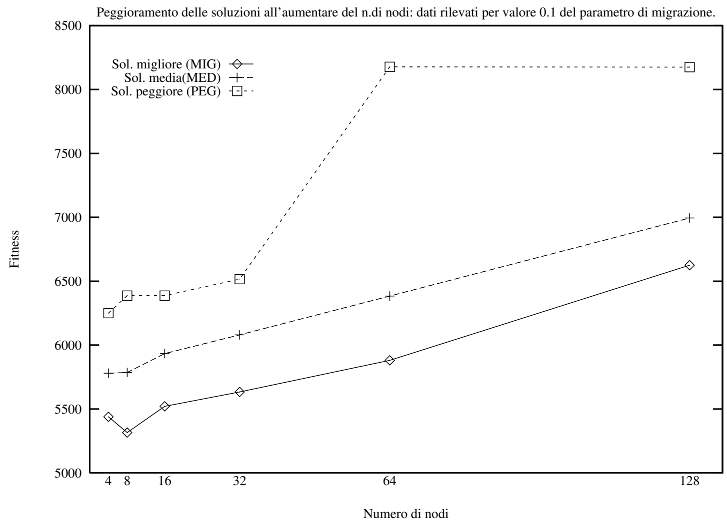


Figura 5.13: Valori di MED, MIG e PEG all'aumentare del numero di nodi per l'algoritmo genetico a *grana grossa* applicato al TSP a 48 città.

prestazione ed efficienza ottenuti elaborando l'algoritmo genetico parallelo all'aumentare il numero dei nodi usati. I tempi medi di esecuzione sono espressi in secondi. Dai dati riportati in Tabella 5.7 e dalla Figura 5.14 si può osservare che si ottiene uno *speedup* superlineare. La causa di questo comportamento è da attribuirsi all'algoritmo di *quicksort* utilizzato per l'ordinamento della popolazione. Infatti dalla Tabella 5.8 si può osservare come lo *speedup* aumenti quasi linearmente per quanto riguarda il tempo richiesto dagli operatori genetici, dal rimpiazzamento e dalla migrazione, mentre abbia un andamento superlineare nel caso del tempo richiesto dall'ordinamento.

Modello a grana fine

Come nel modello *a grana grossa*, anche nel modello *a grana fine* il *crossover* adottato è quello a due punti.

L'algoritmo è stato eseguito per 2000 generazioni sul problema TSP a 48 città, e 3000 generazioni sul problema TSP a 105 città. I valori di *fitness*

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	2600	1	1
4	417	6.23	1.565
8	181	14.36	1.795
16	85	50.58	1.911
32	40	65	2.03
64	20	130	2.03
128	10	260	2.03

Tabella 5.7: Valori di prestazione dell'algoritmo genetico a *grana grossa* per il problema TSP a 48 città.

Numero di nodi	Operatori genetici	Migrazione	Rimpiazzamento	Ordinamento
1	$1149{,}396178 \cdot 10^6$	–	$18{,}278957 \cdot 10^6$	$1433{,}141707 \cdot 10^6$
4	$288{,}592508 \cdot 10^6$	$5{,}403786 \cdot 10^6$	$4{,}987470 \cdot 10^6$	$115{,}820445 \cdot 10^6$
8	$144{,}537829 \cdot 10^6$	$1{,}789940 \cdot 10^6$	$2{,}424009 \cdot 10^6$	$33{,}600413 \cdot 10^6$
16	$72{,}055703 \cdot 10^6$	$0{,}613803 \cdot 10^6$	$1{,}399364 \cdot 10^6$	$9{,}482272 \cdot 10^6$
32	$36{,}455489 \cdot 10^6$	$0{,}210776 \cdot 10^6$	$0{,}668155 \cdot 10^6$	$2{,}983621 \cdot 10^6$
64	$18{,}262981 \cdot 10^6$	$0{,}177746 \cdot 10^6$	$0{,}365765 \cdot 10^6$	$1{,}028947 \cdot 10^6$
128	$9{,}221490 \cdot 10^6$	$0.112435 \cdot 10^6$	$0.207882 \cdot 10^6$	$0.473823 \cdot 10^6$

Tabella 5.8: Tempi (in microsecondi) richiesti dalle principali funzioni dell'algoritmo genetico a *grana grossa* sul problema TSP a 48 città.

raggiunti dall'algoritmo che adotta il modello *a grana fine* sul problema a 48 città sono riportati in Tabella 5.9.

Sul problema a 105 città è stato eseguito l'algoritmo genetico a *grana fine* variando la dimensione dell'ipercubo logico.

In Tabella 5.10 sono riportati i risultati di *fitness* conseguiti eseguendo l'algoritmo a *grana fine* variando le dimensioni della popolazione: si va dai 128 individui corrispondenti ad un ipercubo di dimensione 7, ai 1024 individui corrispondenti ad un ipercubo di dimensione 10. Come era logico aspettarsi, lo schema di *mapping* adottato permette di conseguire *fitness* che migliorano all'aumentare del numero di individui.

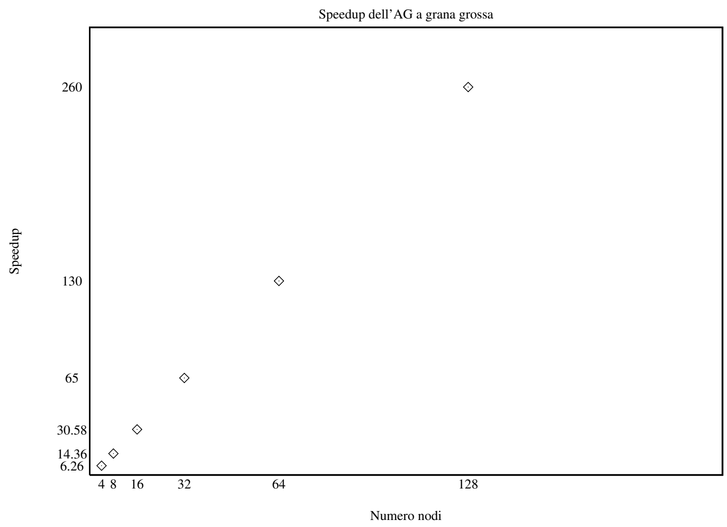


Figura 5.14: Valori di *speedup* dell'algoritmo genetico a *grana grossa* per il TSP a 48 città.

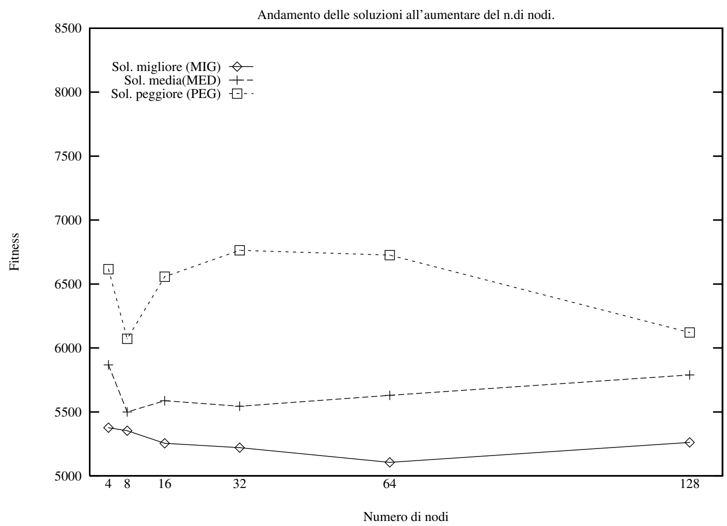


Figura 5.15: Valori di MED, MIG e PEG all'aumentare del numero di nodi per l'algoritmo genetico a *grana fine* eseguito su 128 individui.

	Numero di nodi					
	N=4	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
MED	5868	5499	5588	5544	5630	5789
MIG	5376	5353	5255	5221	5105	5261
PEG	7718	6072	7667	6763	6726	6424

Tabella 5.9: Valori di MED, MIG e PEG dopo 2000 generazioni per l'algoritmo genetico a *grana fine* applicato al TSP a 48 città.

Inoltre, come si può leggere nelle Tabelle 5.11, 5.12, 5.13 e 5.14 relative ai parametri di prestazione dell'algoritmo a *grana fine* eseguito su una popolazione rispettivamente di 128, 256, 512 e 1024 individui, lo *speedup* migliora all'aumentare della popolazione. Fissata la dimensione logica della popolazione lo *speedup* ha un andamento che si avvicina alla linearità, come mostra la Figura 5.16 per la popolazione di 128 individui corrispondente alla dimensione 7 dell'ipercubo. L'algoritmo a *grana fine*, non includendo un algoritmo di ordinamento come l'algoritmo genetico a *grana grossa*, non mostra un andamento superlineare.

Il miglioramento dei valori di *fitness* all'aumentare del numero di nodi è dovuto alle caratteristiche dell'implementazione realizzata: questa tende a minimizzare i tempi di comunicazione a scapito della diversità delle soluzioni allocate sullo stesso nodo. In particolare, il criterio di selezione dei *partner* per l'accoppiamento, tende a scegliere soluzioni all'interno del nodo, allo scopo di minimizzare i tempi di comunicazione tra nodo e nodo. La conseguenza è una maggiore uniformità delle popolazioni allocate nei nodi, che si riflette in valori più alti di *fitness*.

Come mostrato in Tabella 5.15 in cui sono riportati i tempi medi di esecuzione, l'algoritmo genetico a *grana fine* risulta essere scalabile aumentando la dimensione della popolazione e la dimensione dell'ipercubo usato.

		Numero di nodi						
		N=1	*N=4*	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
128 individui	**MED**	39894	24361	23271	23570	23963	22519	22567
	MIG	34207	20774	19830	20532	21230	20269	20593
	PEG	42127	30312	26677	27610	27634	28256	25931
256 individui	**MED**	33375	25313	22146	21616	21695	22247	21187
	MIG	29002	24059	20710	19833	20144	19660	19759
	PEG	40989	26998	23980	24007	23973	24337	22256
512 individui	**MED**	28422	23193	22032	21553	20677	20111	20364
	MIG	28987	22126	19336	20333	19093	18985	18917
	PEG	41020	25684	23450	22807	22213	21696	21647
1024 individui	**MED**	25932	23659	22256	20366	19370	18948	19152
	MIG	27010	21581	21480	18830	18256	18252	17446
	PEG	40901	25307	22757	21714	20766	19525	20661

Tabella 5.10: Valori di *fitness* ottenuti con l'algoritmo genetico a *grana fine* dopo 3000 generazioni al variare del numero di individui assegnati ad ogni nodo per il TSP a 105 città.

Una ulteriore caratteristica degna dell'algoritmo a *grana fine* è che la qualità delle soluzioni non viene degradata al crescere del numero dei processori impiegati, come invece avviene nell'algoritmo genetico *a grana grossa*. Il fatto è chiaramente osservabile confrontando le Figure 5.13, relativa all'algoritmo genetico *a grana grossa*, e 5.15, relativa all'algoritmo genetico *a grana fine* eseguito su 128 individui sul problema a 48 città; la stessa caratteristica si conserva anche per il problema a 105 città.

Nelle Tabelle 5.16 e 5.17 sono riportati, rispettivamente, i valori di *fitness* e i tempi di esecuzione della *grana fine*, in funzione del numero di individui allocati su ogni nodo: come si vede i risultati dipendono dal numero totale di individui allocati, in quanto il processo di *mapping* rende indipendente la struttura della popolazione dal numero di nodi sui quali questa viene allocata.

Ci si potrebbe aspettare che una popolazione di dimensione fissata, esegui-

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	6592	1	1
4	1739	3.79	0.94
8	908	7.25	0.9
16	481	13.7	0.85
32	261	25.25	0.78
64	147	44.84	0.7
128	84	78.47	0.61

Tabella 5.11: Valori di prestazione dell'algoritmo genetico a *grana fine* eseguito su una popolazione di 128 individui per il problema TSP a 105 città.

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	13187	1	1
4	3433	3.84	0.96
8	1764	7.47	0.93
16	921	14.31	0.89
32	488	27.02	0.84
64	266	49.57	0.77
128	149	88.5	0.69

Tabella 5.12: Valori di prestazione dell'algoritmo genetico a *grana fine* eseguito su una popolazione di 256 individui per il problema TSP a 105 città.

ta su uno o più nodi, ottenga valori identici di *fitness*, dato che la struttura della popolazione viene mantenuta dalla tecnica di *mapping* indipendente dal numero di nodi sui quali è eseguito l'algoritmo. Per esempio, si potrebbe pensare che la popolazione di 64 individui (ipercubo logico di dimensione 6, ogni individuo logicamente connesso con 6 vicini), raggiunga identici valori di fitness, sia che venga allocata su 16 nodi a gruppi di 4 individui, sia che venga allocata su 32 nodi a gruppi di 2 individui. Osservando la Tabella 5.16 si rileva come ciò non sia vero; in effetti gli algoritmi genetici implementati differenziano la popolazione iniziale in base al numero di nodi sui quali una

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	26380	1	1
4	6780	3.89	0.97
8	3465	7.61	0.95
16	1784	14.78	0.92
32	932	28.03	0.88
64	495	53.29	0.83
128	269	98.06	0.76

Tabella 5.13: Valori di prestazione dell'algoritmo genetico a *grana fine* eseguito su una popolazione di 512 individui per il problema TSP a 105 città.

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	52770	1	1
4	13441	3.92	0.98
8	6815	7.74	0.96
16	3489	15.12	0.94
32	1796	29.38	0.91
64	940	56.13	0.87
128	501	105.32	0.82

Tabella 5.14: Valori di prestazione dell'algoritmo genetico a *grana fine* eseguito su una popolazione di 1024 individui per il problema TSP a 105 città.

popolazione viene allocata; una popolazione iniziale di 64 individui allocata su 16 nodi è diversa da una popolazione iniziale della stessa dimensione allocata su 8 nodi.

5.4 Confronto tra gli AG e il SA

Per poter effettuare un confronto dei tre algoritmi; AG a *grana fine*, AG a *grana grossa* e algoritmo di SA, si sono confrontati i valori di *fitness* raggiunti da ciascun algoritmo dopo la generazione e la valutazione di 512000 soluzioni. Questo criterio di confronto, anche se il numero di iterazioni non

	Numero di nodi						
	N=1	*N=4*	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
128 individui	6592	1739	908	481	261	147	84
256 individui	13187	3433	1764	921	488	266	149
512 individui	26380	6780	3465	1784	932	495	269
1024 individui	52770	13441	6815	3489	1796	940	501

Tabella 5.15: Tempo di esecuzione dell'algoritmo genetico a *grana fine* dopo 3000 generazioni espressi in secondi, al variare del numero di individui assegnati ad ogni nodo per il TSP a 105 città.

Individui	Numero di nodi			
per nodo	*N=16*	*N=32*	*N=64*	*N=128*
1	27492	25364	24311	22567
2	26327	24521	22519	21187
4	26611	23963	22247	20364
8	23570	21695	20111	19152

Tabella 5.16: Valori di *fitness* medi ottenuti con l'algoritmo genetico a *grana fine* dopo 3000 generazioni al variare del numero di individui assegnati ad ogni nodo per il problema a 105 città.

permette all'algoritmo di SA di convergere all'ottimo, consente di valutare l'"intelligenza" di un algoritmo nella ricerca di buone soluzioni.

L'algoritmo di SA è stato eseguito con i seguenti parametri:

- temperatura iniziale $T_{initial} = 100$;
- temperatura finale $T_{final} = 0.01$;
- massimo numero di iterazioni per temperatura pari a 6000;
- $nuova\_temperatura = 0.98 * temperatura$.

I risultati dell'algoritmo di SA, e degli AG a *grana fine* e a *grana grossa* sono riportati, rispettivamente, nelle Tabelle 5.19, 5.20 e 5.21. Si nota,

Individui	Numero di nodi			
per nodo	*N=16*	*N=32*	*N=64*	*N=128*
1	78	80	83	85
2	139	142	147	149
4	255	261	266	269
8	481	488	495	501

Tabella 5.17: Tempo di esecuzione dell'algoritmo genetico a *grana fine* dopo 3000 generazioni espressi in secondi, al variare del numero di individui assegnati ad ogni nodo per il problema a 105 città.

Numero di nodi	Operatori genetici	Migrazione	Rimpiazzamento
1	$1315,148902 \cdot 10^6$	-	$95,04134 \cdot 10^6$
4	$353,505246 \cdot 10^6$	$9,678888 \cdot 10^6$	$2,341086 \cdot 10^6$
8	$175,037137 \cdot 10^6$	$9,410213 \cdot 10^6$	$1,135749 \cdot 10^6$
16	$87,587880 \cdot 10^6$	$8,420477 \cdot 10^6$	$0,562218 \cdot 10^6$
32	$44,432938 \cdot 10^6$	$6,953502 \cdot 10^6$	$0,289869 \cdot 10^6$
64	$20,826858 \cdot 10^6$	$6,325164 \cdot 10^6$	$0,145998 \cdot 10^6$
128	$10,348201 \cdot 10^6$	$5,743324 \cdot 10^6$	$0,071429 \cdot 10^6$

Tabella 5.18: Tempi (in microsecondi) richiesti dalle principali funzioni dell'algoritmo genetico a *grana fine* sul problema TSP a 48 città.

come era prevedibile, che l'algoritmo di SA, per quanto riguarda la bontà delle soluzioni trovate, si comporta meglio di entrambi gli AG. All'aumentare del numero di nodi, l'algoritmo genetico a *grana fine* si comporta meglio dell'algoritmo genetico a *grana grossa*.

Dal punto di vista dei tempi di esecuzione, utilizzando fino a 8 nodi, l'algoritmo di SA mostra un comportamento migliore degli AG. Questi ultimi, però, all'aumentare del numero di nodi (oltre 16), richiedono meno tempo di esecuzione dell'algoritmo basato sul SA.

Complessivamente l'algoritmo che mostra un miglior compromesso tra bontà delle soluzioni trovate e tempo impiegato per raggiungerle è l'algoritmo genetico a *grana fine*.

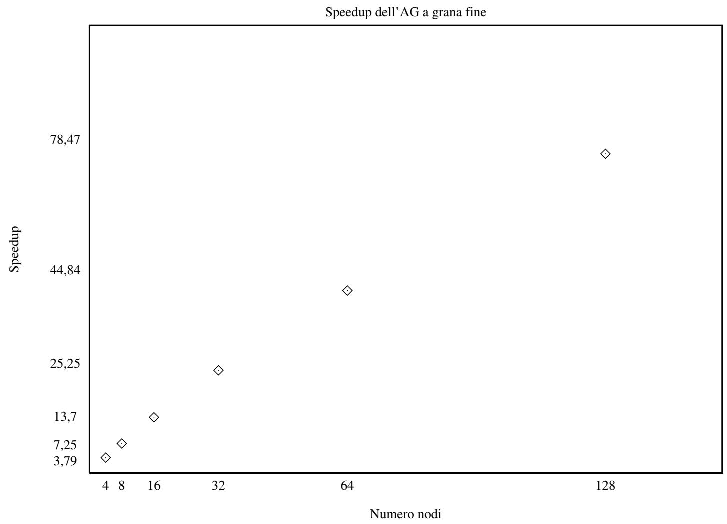


Figura 5.16: *Speedup* per l'algoritmo genetico a *grana fine* eseguito su 128 individui.

	Numero di nodi					
	N=4	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
MED	20000	20450	20698	20724	21744	23102
MIG	18552	19520	19436	18205	19728	20593
PEG	21970	21782	23600	22441	23931	25831
Tempo	565	433	381	356	342	331

Tabella 5.19: Tempi di esecuzione (in secondi) e valori di MED, MIG e PEG per l'algoritmo di SA dopo 512000 valutazioni di soluzioni.

5.5 Conclusioni

In questo Capitolo sono stati presentati i risultati dello studio effettuato sull'applicazione degli AG al problema del commesso viaggiatore. Sono stati dapprima implementati e provati AG sequenziali, allo scopo di effettuare uno studio preliminare sui diversi criteri di rimpiazzamento degli individui all'interno delle popolazioni, su diversi valori del parametro di crossover e sul tipo di *crossover*. Dai risultati ottenuti si è visto che il *crossover* a due

	Numero di nodi					
	N=4	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
MED	26373	26349	25922	25992	24613	23227
MIG	23979	23906	23173	21992	22145	20669
PEG	30140	27851	29237	29193	30176	26801
Tempo	1160	606	321	174	98	51

Tabella 5.20: Tempi di esecuzione (in secondi) e valori di MED, MIG e PEG per l'algoritmo genetico a *grana fine* dopo 512000 valutazioni di soluzioni.

	Numero di nodi					
	N=4	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
MED	23860	24526	27063	29422	32542	35342
MIG	20219	21120	23510	25783	30927	33015
PEG	25299	29348	36795	39330	39131	41508
Tempo	1670	804	392	196	97	47

Tabella 5.21: Tempi di esecuzione (in secondi) e valori di MED, MIG e PEG per l'algoritmo genetico a *grana grossa* dopo 512000 valutazioni di soluzioni.

punti trova soluzioni con *fitness* migliori rispetto al *crossover* ad un punto. La scelta del tipo di crossover è quindi un parametro importante negli AG.

Sono stati quindi implementati, su un'architettura parallela con topologia di interconnessione ad ipercubo, AG paralleli a *grana fine* e a *grana grossa*, e sono stati valutati analizzandone le prestazioni al variare di parametro di migrazione. Per l'algoritmo genetico a *grana grossa* si sono ottenute soluzioni di fitness peggiori all'aumentare del numero di nodi, e uno *speedup* superlineare dovuto alla procedura di ordinamento utilizzata.

Per l'algoritmo genetico a *grana fine* si è adottata una tecnica di *mapping* della popolazione sui processori che ha permesso di rendere indipendente la dimensione della popolazione dal numero di nodi nell'esecuzione dell'algoritmo stesso. Tale tecnica di *mapping* ha permesso di conseguire *fitness* migliori all'aumentare della dimensione della popolazione con un incremento del valore di *speedup* al crescere del numero di individui dela popolazione.

Infine è stato effettuato un confronto tra gli algoritmi di AG paralleli a *grana fine* e *grana grossa* con l'algoritmo di SA presentato in [46] e modificato opportunamente per trattare il problema TSP.

Capitolo 6

Un algoritmo ibrido: AGSA

In questo Capitolo viene presentato un algoritmo ibrido che combina le caratteristiche degli algoritmi genetici a *grana fine* e dell'algoritmo di SA. L'algoritmo genetico a *grana fine* impiegato è quello presentato nel precedente Capitolo.

L'idea di impiegare l'algoritmo genetico a *grana fine* combinandolo con il SA viene dalle considerazioni che seguono.

Un importante parametro degli AG è il numero di soluzioni considerate ad ogni generazione; una popolazione di grandi dimensioni riesce di solito a trovare soluzioni di *fitness* più bassa rispetto a popolazioni di piccole dimensioni.

Consultando le Tabelle 6.1 e 6.2 riportate dal Capitolo precedente per comodità di consultazione, si vede che l'algoritmo genetico a *grana grossa*, eseguito con il parametro di migrazione M=0.3, trova soluzioni migliori dell'algoritmo genetico a *grana fine* quando la popolazione viene partizionata su pochi nodi, per esempio quando si esegue l'algoritmo genetico a *grana grossa* su 4 nodi. Per un numero di nodi maggiore di 8, l'algoritmo genetico a *grana grossa* mostra un peggioramento della bontà delle soluzioni dovuto, come già esposto nel Capitolo precedente, al limitato numero di individui assegnato a ciascun nodo. L'algoritmo genetico a *grana fine* non mostra un peggioramento della qualità delle soluzioni grazie alla strutturazione della popolazione che viene mantenuta identica indipendentemente dal numero di nodi utilizzati.

		Numero di nodi					
		N=4	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
M=0.1	**MED**	23790	24538	27281	29364	32632	37872
	MIG	21370	21829	24532	26517	28966	36021
	PEG	26706	28381	37436	37008	37779	44048
M=0.3	**MED**	23560	24526	27063	29422	32542	35342
	MIG	20219	21120	23510	25783	30927	33015
	PEG	25899	29348	36795	39330	39131	41508
M=0.5	**MED**	23790	24670	26752	29137	32208	35067
	MIG	21272	21411	24088	23373	30617	31603
	PEG	26706	30875	36080	37779	36649	43494

Tabella 6.1: Valori di *fitness* ottenuti con l'AG a *grana grossa* al variare del parametro di migrazione per il TSP a 105 città

Come si può osservare dalla Tabella 6.2 l'algoritmo genetico a *grana fine* riesce a trovare soluzioni migliori all'aumentare della dimensione dell'*ipercubo* logico, ossia del numero di soluzioni trattate. Per contro, l'algoritmo genetico a *grana fine* impiega un tempo maggiore dell'algoritmo genetico a *grana grossa* per trattare un grande numero di individui, a causa dei calcoli necessari a gestire la struttura della popolazione che è invece assente nel modello a *grana grossa*. Quindi l'algoritmo genetico a *grana fine*, per avere tempi di esecuzione simili agli AG a *grana grossa*, deve trattare un minor numero di individui; la minore dimensione della popolazione di soluzioni impiegata ha come conseguenza una esplorazione parziale dello spazio delle soluzioni.

È perciò importante che le soluzioni dell'algoritmo genetico a *grana fine* siano distribuite il più uniformemente possibile sullo spazio delle soluzioni, in

		Numero di nodi						
		N=1	*N=4*	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
128 individui	**MED**	39894	24361	23271	23570	23963	22519	22567
	MIG	34207	20774	19830	20532	21230	20269	20593
	PEG	42127	30312	26677	27610	27634	28256	25931
256 individui	**MED**	33375	25313	22146	21616	21695	22247	21187
	MIG	29002	24059	20710	19833	20144	19660	19759
	PEG	40989	26998	23980	24007	23973	24337	22256
512 individui	**MED**	28422	23193	22032	21553	20677	20111	20364
	MIG	28987	22126	19336	20333	19093	18985	18917
	PEG	41020	25684	23450	22807	22213	21696	21647
1024 individui	**MED**	25932	23659	22256	20366	19370	18948	19152
	MIG	27010	21581	21480	18830	18256	18252	17446
	PEG	40901	25307	22757	21714	20766	19525	20661

Tabella 6.2: Valori di *fitness* ottenuti con l'AG a *grana fine* dopo 3000 generazioni al variare del numero di individui assegnati ad ogni nodo per il TSP a 105 città.

modo da poter considerare il maggior numero possibile di *regioni* dove possono essere situati gli ottimi locali. Per aumentare l'uniformità della distribuzione delle soluzioni dell'algoritmo genetico a *grana fine* sullo spazio delle soluzioni, si può impiegare una procedura di SA che opera ad ogni generazione dell'algoritmo genetico sulla soluzione corrente nel seguente modo:

- inizialmente, quando la temperatura dell'algoritmo di SA è alta, accetta anche soluzioni che peggiorano la *fitness* dell'individuo trattato;
- verso la fine della ricerca delle soluzioni, accetta solo soluzioni che migliorano la *fitness* dell'individuo, e quindi realizza una procedura di *ricerca locale*.

Quindi, grazie al criterio di accettazione di nuove soluzioni dell'algoritmo di SA, si ottiene una maggiore uniformità iniziale della distribuzione degli individui nello spazio delle soluzioni, e si ha il posizionamento di individui anche in *regioni* apparentemente non indicate per la ricerca di soluzioni di bassa *fitness*; in seguito l'abbassamento della temperatura consente di scoprire ottimi locali situati in *regioni* che, con la normale procedura degli AG, sarebbero stati esplorati con minore probabilità.

Nelle Figure 6.1 e 6.2 sono riportati rispettivamente gli pseudocodici degli algoritmi AGSA e della procedura di SA utilizzata dall'algoritmo AGSA.

Come si vede dalle Figure, l'algoritmo AGSA è un AG a *grana fine* con una chiamata alla procedura Simulated Annealing; inoltre la gestione dell'abbassamento della temperatura c, contrariamente a quanto avviene nell'algoritmo di SA, non è effettuata dalla procedura di SA, ma dall'algoritmo AGSA. Ciò consente di utilizzare la procedura di Simulated Annealing alla temperatura attuale per un numero prestabilito di iterazioni. Dopo aver invocato la procedura Simulated Annealing, l'algoritmo AGSA effettua il controllo sulla temperatura raggiunta: quest'ultima viene abbassata solamente se non si è ancora raggiunta la temperatura finale c_f. L'algoritmo AGSA prosegue poi con il normale funzionamento dell'algoritmo genetico a *grana fine*.

La procedura di SA, il cui pseudocodice appare in Figura 6.2, riceve come parametro dall'algoritmo AGSA la temperatura attuale c, e si limita ad eseguire sulla soluzione corrente un numero prestabilito di iterazioni, accettando o rifiutando le modifiche alla soluzione corrente secondo il normale criterio dell'algoritmo di SA. Inizialmente, quando la temperatura è alta, saranno accettate anche soluzioni che forniscono valori di *fitness* più alti: questo è il meccanismo che consente di diffondere la popolazione su tutto lo spazio delle soluzioni. In seguito, una volta raggiunta la temperatura finale (c_f), verranno accettate solamente soluzioni che migliorano la *fitness*.

```
Program AGSA;
whoami(mionodo,M);
/*Determina l'identificatore del nodo e la dimensione dell'ipercubo */
TOTALE_INDIVIDUI = 2^N;
num_processori = 2^M;
num_individui = TOTALE_INDIVIDUI div num_processori;
popolazione_iniziale=GENERA();
ASSEGNA_POSIZIONE(popolazione_iniziale);
MAPPING(popolazione_iniziale,num_individui,mionodo);
/*Assegna su ogni nodo num_individui di elementi vicini su nodi allocati vicini fisicamente*/
c = c0;
/*Inizializza la temperatura c al valore iniziale c0 */
while condizione_di_terminazione do
begin

        INVIA_A_NODI_VICINI();
        RICEVI_DA_NODI_VICINI();
        for cella = 1 to num_individui do
        begin

                contenuto_cella=SELEZIONA(popolazione);
                /* Seleziona l'individuo tra quelli assegnati a mionodo */
                contenuto_cella=SIMULATED_ANNEALING(contenuto_cella,c);
                COMPUTA_FITNESS(contenuto_cella);
                /*Richiama la procedura SA, che opera per un numero fissato di iterazioni;
                poi abbassa la temperatura, fino ad arrivare alla temperatura finale cf*/
                individuo_vicino=SELCASUALE(popolazione);
                /* Seleziona un individuo vicino tra gli individui che sono nell'intorno*/
                CROSSOVER(individuo_vicino,contenuto_cella,figlio1,figlio2);
                /* Genera due figli applicando il crossover su individuo_vicino e
                contenuto_cella */
                figlio_da_mutare =SELFIGLIO(figlio1,figlio2);
                /*Seleziona uno dei due figli per applicare la mutazione*/
                MUTAZIONE(figlio_da_mutare);
                COMPUTA_FITNESS(figlio1,figlio2);
                /* Calcola i valori di fitness dei figli generati */
                contenuto_cella= RIMPIAZZAMENTO(contenuto_cella,figlio1,figlio2);
                /*Seleziona l'individuo che deve occupare la posizione cella: viene selezionato
                l'individuo con valore di fitness più basso*/

        end;
        if (c > cf) then c = f(c);


end;
end
```

Figura 6.1: Algoritmo AGSA espresso in pseudocodice.

```
Procedure Simulated_Annealing(contenuto_cella, c);
repeat

        costo = CALCOLA_FITNESS(contenuto_cella);
        nuova_soluzione = PERTURBA(contenuto_cella);
        nuovo_costo = CALCOLA_FITNESS(nuova_soluzione);
        Δ(C) = nuovo_costo - costo;
        if (Δ(C) < 0) | ((e^(Δ(C)/c)) > random (0,1)) then
        begin

               contenuto_cella = nuova_soluzione;
               costo = nuovo_costo;

        end


until (numero_di_iterazioni);
end
```

Figura 6.2: Procedura di Simulated Annealing impiegata nell'algoritmo AGSA.

6.1 Risultati sperimentali

L'algoritmo AGSA è stato eseguito con il seguente programma di raffreddamento:

- temperatura iniziale $T_{initial} = 100$;
- temperatura finale $T_{final} = 0.01$;
- numero di iterazioni della procedura di Simulated Annealing pari a 200;
- $nuova\_temperatura = 0.95 * temperatura$.

Il problema TSP usato per le prove sull'algoritmo AGSA è il TSP a 105 città. Sono state sufficienti 1000 generazioni per arrivare al punto di convergenza dell'algoritmo, oltre il quale esso non riesce a migliorare ulteriormente le soluzioni.

In Tabella 6.3 sono riportati i risultati delle esecuzioni dell'algoritmo AGSA: i risultati rilevati sono la soluzione migliore (**MIG**), peggiore (**PEG**) e media (**MED**) trovate.

		Numero di nodi						
		N=1	*N=4*	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
128 individui	**MED**	20456	19069	19865	19002	18997	19018	18057
	MIG	18715	18513	19175	17977	17864	18527	16894
	PEG	20593	19316	20494	19780	19515	19776	19072
256 individui	**MED**	18563	18918	18859	18498	18554	18431	18677
	MIG	18071	17850	18297	18087	17372	17513	17813
	PEG	19946	19858	19814	18929	19638	18843	19538
512 individui	**MED**	18504	18755	18070	18221	17879	17956	18066
	MIG	18122	17941	17302	17520	17048	16733	17594
	PEG	19847	19503	18505	19093	18724	19085	18515
1024 individui	**MED**	18497	18296	17697	17763	17260	17960	17929
	MIG	18911	18962	17140	17283	16469	16961	16963
	PEG	19115	19005	18126	18284	17918	18676	18885

Tabella 6.3: Valori di *fitness* ottenuti con l'algoritmo AGSA dopo 3000 generazioni al variare del numero di individui assegnati ad ogni nodo per il TSP a 105 città

Come si rileva dalla lettura della Tabella appena citata e dalla Figura 6.3, l'algoritmo AGSA non degrada la qualità delle soluzioni all'aumentare del numero di nodi.

Una delle caratteristiche più importanti dell'algoritmo genetico a *grana fine* viene quindi conservata nell'algoritmo AGSA. Inoltre si nota dalla Figura 6.3 come l'algoritmo AGSA riesca a migliorare la qualità media delle soluzioni, riuscendo nel contempo a diminuire, rispetto all'algoritmo a *grana fine*, la differenza tra la soluzione migliore (MIG) e la peggiore (PEG) trovata.

Confrontando le Tabelle 6.2 e 6.4 relative all'algoritmo a *grana fine* del precedente Capitolo, qui riportate per comodità di lettura, con le Tabelle 6.5 e 6.3, relative all'algoritmo AGSA, si nota che l'algoritmo AGSA, già con una popolazione di soli 128 individui, riesce ad ottenere prestazioni migliori, in termini di fitness, dell'algoritmo genetico a *grana fine* eseguito su una

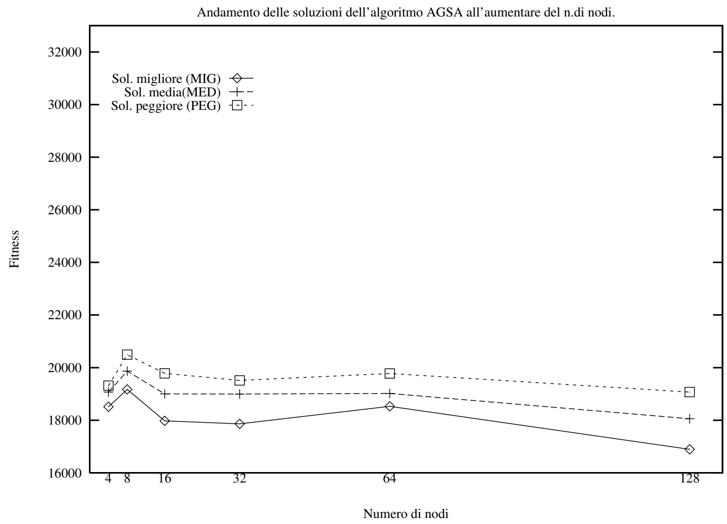


Figura 6.3: Andamento di MED, MIG e PEG all'aumentare del numero di nodi per l'algoritmo AGSA eseguito con 128 individui.

popolazione di 1024 individui. Anche per quanto riguarda i tempi di esecuzione, l'algoritmo AGSA ottiene migliori prestazioni rispetto all'algoritmo a *grana fine*.

	Numero di nodi						
	N=1	*N=4*	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
128 individui	6592	1739	908	481	261	147	84
256 individui	13187	3433	1764	921	488	266	149
512 individui	26380	6780	3465	1784	932	495	269
1024 individui	52770	13441	6815	3489	1796	940	501

Tabella 6.4: Tempo di esecuzione dell'AG a *grana fine* dopo 3000 generazioni espressi in secondi, al variare del numero di individui assegnati ad ogni nodo per il TSP a 105 città

In Tabella 6.6 sono riportati i tempi medi di esecuzione e lo *speedup* dell'algoritmo AGSA ottenuti eseguendo l'algoritmo con una popolazione di 128 individui. I tempi medi di esecuzione sono espressi in secondi.

	Numero di nodi						
	N=1	*N=4*	*N=8*	*N=16*	*N=32*	*N=64*	*N=128*
128 individui	4775	1224	624	321	167	89	48
256 individui	9544	2430	1232	629	324	169	90
512 individui	19085	4833	2442	1238	632	326	170
1024 individui	38050	9628	4849	2450	1243	635	328

Tabella 6.5: Tempo di esecuzione dell'algoritmo AGSA dopo 1000 generazioni espressi in secondi, al variare del numero di individui assegnati ad ogni nodo per il TSP a 105 città

Numero di nodi	Tempo (in sec.)	Speedup	Efficienza
1	4775	1	1
4	1224	3.9	0.97
8	624	7.65	0.95
16	321	14.87	0.92
32	167	28.59	0.89
64	89	53.65	0.83
128	48	99.47	0.77

Tabella 6.6: Valori di prestazione dell'algoritmo AGSA eseguito con 128 individui per il problema TSP a 105 città.

Dai dati riportati in Tabella 6.6 e dalla Figura 6.4 si osserva che lo *speedup* rimane molto vicino alla linearità.

Si può notare inoltre come lo *speedup* dell'algoritmo AGSA risulti superiore a quello dell'algoritmo a *grana fine*.

6.2 Conclusioni

In questo Capitolo è stato presentato un algoritmo ibrido che combina le caratteristiche degli algoritmi genetici a *grana fine* e dell'algoritmo di SA. L'algoritmo genetico a *grana fine*, data la struttura della popolazione che gestisce, non riesce a trattare grandi quantità di soluzioni in tempi parago-

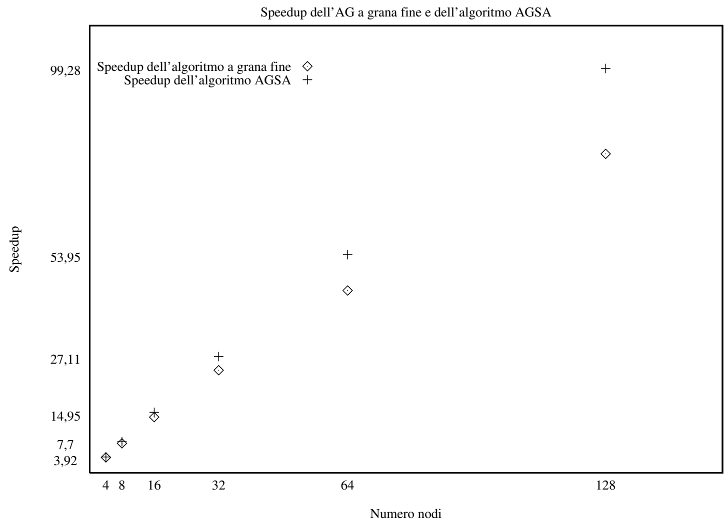


Figura 6.4: *Speedup* per gli algoritmi a grana fine e AGSA eseguiti con 128 individui.

nabili a quelli mostrati dall'algoritmo genetico a *grana grossa*; la minore dimensione della popolazione di soluzioni impiegata ha come conseguenza una esplorazione parziale dello spazio delle soluzioni.

È perciò importante che le soluzioni dell'algoritmo genetico a *grana fine* siano distribuite il più uniformemente possibile sullo spazio delle soluzioni, in modo da poter considerare il maggior numero possibile di *regioni* dove possono essere situati gli ottimi locali.

Per aumentare l'uniformità della distribuzione delle soluzioni dell'algoritmo genetico a *grana fine* sullo spazio delle soluzioni, è stata impiegata una procedura di SA che, all'inizio del processo di ricerca, posiziona individui della popolazione anche in *regioni* apparentemente non indicate per la ricerca di soluzioni di bassa *fitness*.

L'algoritmo AGSA ha mostrato di convergere in sole 1000 generazioni a soluzioni di qualità superiore a quelle ottenute con l'algoritmo genetico a *grana fine* eseguito per 3000 generazioni, mostrando nel contempo minori tempi di esecuzione ed uno *speedup* maggiore.

Capitolo 7

Conclusioni

Nel lavoro di tesi sono stati presi in considerazione algoritmi di approssimazione per la soluzione di problemi di ottimizzazione combinatoriale. Il ricorso ad algoritmi di approssimazione con complessità polinomiale è infatti indispensabile quando la dimensione dei problemi trattati diviene significativa. Anche gli algoritmi euristici sono tuttavia caratterizzati da complessità computazionali elevate che rendono spesso difficile trovare buone soluzioni in tempi ragionevoli. Per ovviare a questa limitazione, che spesso costringe a sacrificare la qualità della soluzione a scapito di tempi di esecuzione accettabili, sono state progettate, implementate e valutate versioni parallele di *algoritmi genetici* (AG) a *grana fine* e a *grana grossa*.

E' stato utilizzato un *multicomputer* nCUBE2 con 128 nodi di elaborazione, e gli algoritmi di approssimazione implementati sono state applicati alla risoluzione del problema del commesso viaggiatore che per la sua semplicità formale ben si presta ad essere usato come *benchmark* per la valutazione di euristiche. Per permettere confronti con altre euristiche, è stato modificato opportunamente per adattarlo al *benchmark* scelto l'algoritmo di *Simulated Annealing* presentato in [46].

Per alcune implementazioni di AG sono state progettate ed implementate strategie originali. In particolare per l'algoritmo genetico a *grana fine* presentato nel Capitolo 5, al fine di ovviare in maniera semplice ed efficiente alla carenza di buona parte delle implementazioni del modello a *grana fine*, è stato progettato uno schema di *mapping* ottimo che, insieme ad una virtualizzazione del funzionamento dell'algoritmo, rende il numero di individui della popolazione indipendente rispetto al numero di processori fisici disponibili. I risultati delle sperimentazioni di questo algoritmo hanno mostrato come all'aumentare della popolazione si ottengono soluzioni con valori di *fitness* migliori.

È stato inoltre implementato un nuovo algoritmo che combina le caratteristiche migliori degli algoritmi genetici a *grana fine* e del *Simulated Annealing*. L'algoritmo, denominato AGSA, ha mostrato un comportamento migliore dell'algoritmo a *grana fine*, riuscendo a conseguire *fitness* di valore inferiore in minor tempo di esecuzione.

La scelta degli operatori genetici, dei criteri di rimpiazzamento degli individui all'interno delle popolazioni, dei valori del parametro di *crossover* sono state effettuate sulla base di *test* approfonditi che hanno evidenziato quali operatori e quali parametri conducessero ad una migliore convergenza dell'algoritmo per il problema del commesso viaggiatore.

Tali risultati sono stati conseguiti dopo uno studio approfondito della letteratura esistente, documentato dai primi quattro capitoli della tesi. Sono stati dapprima esaminati *algoritmi genetici* sequenziali ed i loro *operatori genetici*: *selezione*, *crossover*, *mutazione*, *inversione*. Sono stati esposti il modello di generazione discreto e quello continuo. Sono stati inoltre trattati aspetti teorici riguardanti gli *algoritmi genetici* tratti da [23] e da [14]. È stato presentato un risultato fondamentale, noto come *Teorema degli schemi*.

Si è passato quindi ad esporre alcuni modelli di parallelizzazione degli *algoritmi genetici* su elaboratori paralleli:

- a *grana fine*: si opera su una singola popolazione in cui ogni individuo è posizionato in una cella della topologia della struttura di interconnessione usata. Gli operatori genetici vengono applicati solamente tra individui vicini;
- a *grana grossa*: varie sottopopolazioni si evolvono in parallelo, scambiandosi periodicamente gli individui che forniscono soluzioni migliori;
- *centralizzata*: una popolazione *panmitica* è elaborata da vari processi, alcuni dei quali, i processi *slave*, realizzano gli operatori genetici classici, mentre un solo processo, chiamato *master*, seleziona ed invia agli *slave* gli individui che forniscono soluzioni migliori.

Si sono quindi descritti alcuni esempi di modelli di parallelizzazione a *grana fine*, a *grana grossa* e *centralizzata* esistenti in letteratura.

È stato poi presentato il classico algoritmo di *Simulated Annealing*, discutendo la scelta dei parametri che governano il funzionamento dell'algoritmo, e che sono:

- il valore iniziale del parametro di controllo (c_0);
- il valore finale del parametro di controllo (c_f), che definisce il *criterio di arresto* dell'algoritmo;
- la regola che stabilisce come far variare il parametro di controllo c_k dall'iterazione corrente k, alla successiva $(k + 1)$, detto *criterio di decremento*.

Si è esaminato un algoritmo *ibrido* che combina l'algoritmo di *Simulated Annealing* con tecniche di *ricerca locale* ([35]).

Sono state presentate alcune soluzioni al problema di parallelizzare l'algoritmo di *Simulated Annealing* distinguendo, tra gli altri, due approcci:

- criterio a *decomposizione di mossa*;
- criterio a *mosse parallele*.

È stato esposto un metodo ([46]) che sfrutta la tecnica del calcolo speculativo, ed un altro ([3]) che usa una architettura $Worker - Farmer$.

Ulteriori sviluppi futuri del lavoro di tesi potrebbero essere i seguenti:

- confronto dell'algoritmo AGSA con un algoritmo genetico a *grana fine* che includa una procedura di ricerca locale;
- applicazione dell'algoritmo AGSA a differenti problemi di ottimizzazione combinatoriale, allo scopo di verificarne il comportamento anche su altri *benchmark*.

Appendice A

Architettura di NCube

A.1 Architettura del sistema nCUBE2

Il sistema nCUBE2 utilizza dei chip custom VLSI a 25 MHZ ed è un *multicomputer* con architettura distribuita, che prevede l'utilizzo di al massimo 8196 (2^{13}) nodi di elaborazione, connessi mediante una rete con topologia ad ipercubo binario (vedi Figura A.1). NCUBE 2 necessita di un *host computer*, di solito una workstation Sun o Silicon Graphics.

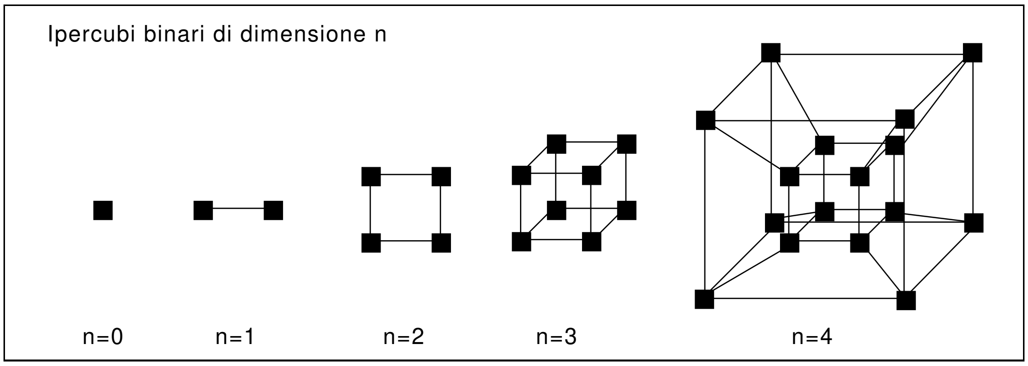


Figura A.1: Topologia dell'ipercubo binario in funzione del numero di dimensioni.

A.1.1 Nodo di elaborazione

Ogni nodo di elaborazione ha al suo interno un processore custom VLSI, una memoria privata ed un insieme di 14 canali di comunicazione bidirezionali. Di questi canali, 13 servono per le comunicazioni inter–nodo ed 1 è dedicato per comunicare con l'host e con il sottoinsieme di I/O.

Uno schema dell'architettura del nodo di elaborazione e del processore è descritto in Figura A.2. [44]

Ogni processore di nCUBE2 è in grado di indirizzare da 1 a 64 MByte di memoria. Gli indirizzi di memoria sono di 32 bit, più 7 bit di ECC. I 28 canali unidirezionali comunicano senza che sia necessario l'intervento della CPU (tranne che per la fase di inizializzazione della comunicazione), utilizzando routing dimensionale con controllo di flusso cut–through implementato ad harware. I canali di comunicazione operano in DMA ed ognuno di essi utilizza dei registri privati.

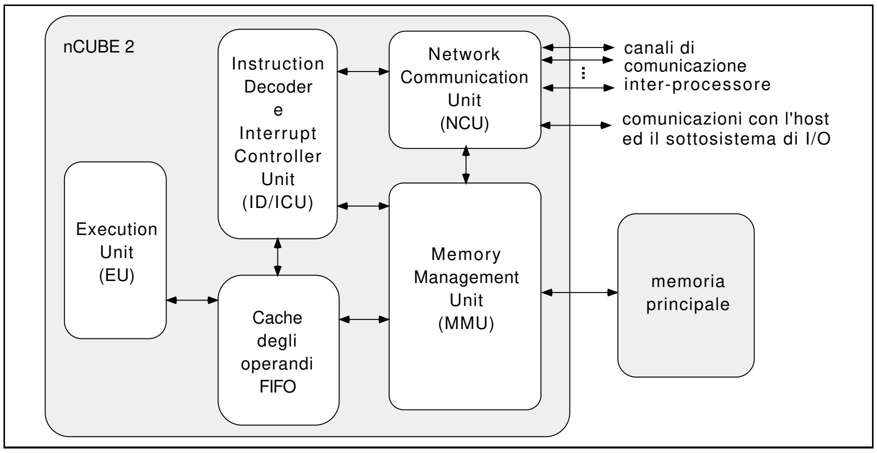


Figura A.2: Architettura del nodo.

Le componenti che costituiscono il processore sono le seguenti:

- *unità di decodifica delle istruzioni e controllo delle interruzioni*

(ID/ICU); si occupa di decodificare le istruzioni e di gestire le interruzioni;

- *unità d'esecuzione (EU);* elabora le operazioni in virgola fissa ed in virgola mobile;
- *unità di comunicazione con la rete (NCU);* si occupa della ricezione e dell'invio dei messaggi dai e ai canali inter–nodo , effettua le comunicazioni con l'host computer e con il sottosistema di I/O ed esegue le operazioni di routing dei messaggi in transito verso altri nodi;
- *unità di gestione della memoria (MMU):* effettua le operazioni di traduzione degli indirizzi e si occupa dei controlli di protezione negli accessi in memoria;
- *cache degli operandi;* memorizza gli operandi per l'esecuzione delle istruzioni. Utilizza la tecnica FIFO per la gestione dei dati.

A.1.2 Sottosistema di comunicazione

Le comunicazioni tra processi sono effettuate tramite lo *scambio di messaggi.* Il processore che intende inviare un messaggio (send) utilizza l'operazione *nwrite*. La trasmissione di un messaggio ha luogo mediante tre fasi :

- individuazione del cammino tra nodo mittente e nodo destinatario;

- trasmissione del messaggio vera e propria;

- restituzione dei canali riservati per la comunicazione.

Per prima cosa i dati vengono copiati nel buffer dei messaggi del nodo mittente. Successivamente il sisitema operativo trasmette il messaggio attraverso i canali della rete, i quali hanno una banda di comunicazione di 2.2

MByte/sec. La trasmissione è *asincrona*, per cui il processore mittente può continuare l'elaborazione non appena ha copiato il messaggio nel buffer. Il messaggio presente nel buffer del nodo destinatario, può essere copiato in memoria tramite l'istruzione *nread* (receive). La *nread* è *bloccante*; dopo la chiamata il processore destinatario attende fino a quando il messaggio è disponibile. A parte un breve tempo necessario all'inizializzazione della comunicazione, l'invio del messaggio ed il resto dell'elaborazione possono avvenire in parallelo.

Il meccanismo di trasmissione prevede che un messaggio sia suddiviso in *pacchetti* di cui il primo contiene l'indirizzo del destinatario, che è utilizzato dai nodi attraverso i quali può transitare il messaggio per determinare il successivo canale mediante il quale si dovrà trasmettere. Durante il trasferimento dal mittente al destinatario, i canali che sono attraversati dal primo pacchetto del messaggio, sono riservati ai restanti pacchetti che compongono il messaggio. La NCU (*Network Communication Unit*) è in grado di bufferizzare due pacchetti per canale. Non appena la NCU ha spazio sufficiente, richiede un altro pacchetto alla NCU del nodo precedente nel cammino percorso dal messaggio. Quando transita il messaggio EOT (End Of Transmission), che segnala la fine della trasmissione, i canali sono rilasciati in modo che siano utilizzabili per altre comunicazioni. Un messaggio formato da pochi pacchetti può essere memorizzato interamente nei buffer dei nodi intermedi prima che il pacchetto iniziale raggiunga il destinatario. Qualsiasi errore nella comunicazione, rilevato dai nodi intermedi, viene codificato nei restanti pacchetti del messaggio ed in questo modo portato a conoscenza del nodo destinatario.

L'architettura della NCU è suddivisa in tre livelli:

- **livello d'interconnessione**

- **livello di routing**
- **livello dei messaggi**

Il **livello d'interconnessione** fornisce l'hardware necessario affinché la comunicazione abbia luogo.

Uno dei canali bidirezionali è dedicato alle comunicazioni con il sottosistema di I/O e con l'host computer. Questo canale si collega con un nodo che fa da interfaccia con l'host e con un insieme di nodi dedicati alla gestione del file system. I canali operano indipendentemente dalla CPU, permettendo che le comunicazioni avvengano parallelamente alla computazione.

Il **livello di routing** gestisce la creazione, il mantenimento e la rimozione dei cammini per lo scambio dei messaggi tra i processori della rete. Per svolgere i suoi compiti deve conoscere l'esatta configurazione del sistema. Il livello di routing si basa sul fatto che, in fase di inizializzazione del sistema, ad ogni nodo è associato un identificatore univoco che lo contraddistingue. La rappresentazione in codice binario dell'indirizzo di due nodi adicenti differisce di una cifra; se i nodi sono collegati tramite il canale n, differisce solo nel valore della cifra n. Il cammino tra mittente e destinatario è stabilito tramite l'invio di un pacchetto. Ogni nodo confronta il contenuto del pacchetto con il proprio identificatore, e spedisce il pacchetto tramite il canale il cui numero corrisponde alla posizione del bit meno significativo in cui i due valori differiscono. I nodi inviano i messaggi tramite un canale il cui indirizzo è maggiore o uguale all'indirizzo del canale sul quale lo hanno ricevuto.

Supponiamo, per esempio, di avere un sistema con 16 nodi connessi mediante un ipercubo binario a 4 dimensioni e che il nodo 0000 debba inviare un messaggio al nodo 1011. Si avranno i seguenti passi:

1. Il processore del nodo 0000 mette il messaggio nel canale 0, in quanto gli indirizzi del mittente (0000) e del destinatario (1011) differiscono nel bit di posizione 0. Questa parte è implementata a livello software dal sistema operativo del nodo.

2. A livello firmware, la NCU del nodo 0000 invia il messaggio alla NCU del nodo 0001, mediante il canale 0.

3. La NCU del nodo 0001 confronta l'identificatore del suo nodo con l'indirizzo del destinatario del messaggio (1011) e rileva che la prima differenza si ha sul bit di posizione 1; quindi instrada il messaggio sul canale di uscita 1.

4. Il messaggio viene inviato alla NCU del nodo 0011.

5. la NCU del nodo 0011 confronta l'identificatore del suo nodo con l'indirizzo del destinatario del messaggio (1011) e vede che la prima differenza si ha sul bit di posizione 3; quindi instrada il messaggio sul buffer del canale di uscita 3.

6. Il messaggio viene inviato alla NCU del nodo 1011.

7. La NCU del nodo 1001 confronta l'identificatore del suo nodo con l'indirizzo del destinatario del messaggio (1011) e vede che sono uguali, quindi trasferisce il messaggio nel buffer, rendendolo disponibile al processo richiedente.

Oltre alla comunicazione punto a punto, la NCU consente delle comunicazioni di tipo *broadcast*, creando una comunicazione tra un mittente e più destinatari.

Il principale vantaggio del broadcast è che si ha la riduzione sia della banda di memoria utilizzata per la comunicazione che dei costi di inizializzazione delle comunicazioni.

Uno svantaggio del broadcast è dovuto al maggior numero di canali coinvolti nella comunicazione. Mentre il pacchetto contenente l'indirizzo del destinatario si muove attraverso la rete, i canali necessari alla comunicazione vengono riservati e collegati tra di loro a formare il cammino tra mittente e destinatari. Prima che la comunicazione avvenga, è necessario che tutti i canali utilizzati siano disponibili. Più sono i canali coinvolti, più alta è la probabilità che la creazione del cammino venga bloccata dall'attesa di un canale già coinvolto in un'altra comunicazione.

Il **livello dei messaggi** fornisce i servizi per il trasferimento dei dati tra i processori. Quando tutti i pacchetti di un messaggio sono stati trasferiti, il processore genera ed invia un messaggio di EOM (End Of Messagge) o di EOT (End Of Transmition), dipendentemente dal fatto che il messaggio inviato sia o meno l'ultimo della trasmissione. EOM mantiene il cammino tra mittente e destinatario (o destinatari), mentre EOT chiude la comunicazione. Durante una trasmissione può essere spedito un numero arbitrario di messaggi separati da EOM, con l'ultimo messaggio seguito da EOT.

A.1.3 Sottosistema di I/O

Oltre ai canali di comunicazione appena descritti, ogni nodo ha un canale DMA dedicato alla comunicazione con i nodi di I/O. Questo canale opera alla stessa velocità degli altri. I nodi di I/O hanno la stesso processore e la stessa configurazione di memoria dei nodi di elaborazione, con in più hardware e software per l'accesso ai dischi e ad altri dispositivi periferici. Il numero dei processori di I/O è variabile in base alle esigenze degli utenti del sistema.

Il sistema nCUBE2 ha un'architettura shared–disk con tutti i nodi di elaborazione che accedono ad un file system unico. Non necessariamente tutti i nodi di elaborazione sono connessi direttamente al sottosistema di I/O, una parte di essi potrebbe accedere ai dispositivi periferici (ed essere interfacciata con l'host computer) indirettamente, utilizzando dei nodi intermedi. Anche nel caso in cui ogni nodo è collegato ad un processore di I/O, i dischi non sono dedicati al singolo nodo, ma condivisi da tutto il sistema.

A.1.4 Ambiente operativo nCube 2

Su ogni nodo è eseguito il sistema operativo nCX che occupa 4 Kbytes. Esso è preposto al caricamento dei programmi sui nodi, alla gestione delle comunicazioni e dei segnali scambiati tra i nodi e tra questi e l'elaboratore host. Un ulteriore programma di sistema è eseguito sulla scheda PIB (Platform Interface Board) per la gestione delle operazioni di I/O tra *host* e nodi.

Per lo sviluppo delle applicazioni viene usato l'ambiente software previsto dall'elaboratore host con l'aggiunta di due cross-compiler FORTRAN 77 e C, una libreria di funzioni per l'implementazione del parallelismo, ed altri strumenti quali Debugger e Perfomance Monitor. La libreria include funzioni per l'allocazione dell'ipercubo, il caricamento dei moduli eseguibili, l'implementazione delle comunicazioni, la gestione dei *multitasking* e l'implementazione dell'I/O parallelo.

I sistemi nCUBE2 supportano l'accesso multiutente suddividendo l'insieme dei nodi in sotto-ipercubi.

Appendice B

Operatori genetici per il TSP

In questa appendice viene descritta l'applicazione dell'algoritmo genetico al problema del commesso viaggiatore. Più precisamente si dovrebbe parlare di applicazioni di programmazione evolutiva, in quanto la rappresentazione binaria per i percorsi e l'uso degli operatori genetici su stringhe binarie, tipici degli algoritmi genetici tradizionali, non sono adeguati al problema del TSP. Nelle prossime sezioni descriviamo gli operatori dell'algoritmo genetico per il problema del commesso viaggiatore.

L'uso degli operatori di *crossover* e di mutazione su stringhe binarie potrebbe dare origine a percorsi illegali, con città ripetute prima della conclusione del percorso; pertanto l'uso della rappresentazione binaria, degli operatori di *crossover* e di mutazione tradizionali rende necessario l'uso di algoritmi detti riparatori, che risolvono le incongruenze generate dalla manipolazione genetica dei percorsi.

Tutti questi motivi hanno reso prevalente l'uso della rappresentazione ad interi dei percorsi tra le città e l'uso di operatori genetici intelligenti, che inglobano parte della conoscenza specifica del problema, e prevengono la costruzione di percorsi illegali.

Nei paragrafi successivi presentiamo brevemente varie rappresentazioni genetiche relative al TSP e vari operatori genetici applicati specificamente al problema del commesso viaggiatore.

B.1 Rappresentazione genetica

Le rappresentazioni che presentiamo riflettono la struttura del problema.

Abbiamo due tipi di rappresentazioni dovute a [38]:

- rappresentazione a **vettori**;
- rappresentazione a **matrici**.

B.1.1 Rappresentazione a vettori

Nella rappresentazione a *vettori* si distinguono tre tipi di rappresentazione:

- *a percorso* ;
- *ad adiacenza* ;
- *ad ordinali.*

La rappresentazione a *percorso* è quella più intuitiva, in essa il percorso è rappresentato mediante una permutazione di città.

Essa ha lo svantaggio di non avere una rappresentazione unica, cioè ogni percorso ha $2n$ differenti rappresentazioni, dove n è il numero di città nel percorso. Ad esempio la configurazione

$$(517894623)$$

rappresenta il percorso:

$$5- > 1- > 7- > 8- > 9- > 4- > 6- > 2- > 3.$$

Nella rappresentazione ad *adiacenza* un percorso è descritto da una lista di città; cioè se il percorso contine un arco dalla città i alla città j allora la citta j è in posizione i. Ad esempio la configurazione

$$(248397156)$$

rappresenta il percorso:

$$1->2->4->3->8->5->9->6->7.$$

Nella rappresentazione ad *ordinali*, il percorso del commesso viaggiatore tra le città è rappresentato mediante una lista l di n numeri interi, mentre un'altra lista funziona come una lista di riferimento.

Il passaggio dalla lista l al percorso avviene prendendo in sequenza gli elementi della lista di riferimento che si trovano nelle posizioni indicate dalla lista l, e rimuovendo le città dalla lista di riferimento ogni volta che vengono selezionate. Ad esempio avendo la seguente lista di riferimento

$$C = (123456789),$$

il percorso

$$A = 1->2->4->3->8->5->9->6->7$$

è rappresentato con la seguente lista

$$l = (112141311),$$

il percorso si costruisce nel seguente modo:

- il primo numero della lista l è 1, così si prende la prima città della lista C come prima città del percorso e si rimuove quest'ultima dalla lista C ottenendo il percorso parziale 1;

- il successivo elemento della lista l è ancora 1, così si prende il primo elemento della lista C, che è la città 2, lo si rimuove da C e si aggiunge al percorso parziale ottenendo $1- > 2$;
- il successivo elemento della lista l è 2, si prende la seconda città della lista corrente C come successiva città del percorso (città 4) e la si rimuove da C ottenendo il percorso parziale $1- > 2- > 4$;
- procedendo allo stesso modo fino a quando non si è esaminata tutta la lista l e quindi ottenuto il percorso finale A.

Il principale vantaggio di questo tipo di rappresentazione è che, per essa, il *crossover* tradizionale funziona senza generare incongruenze, cioè combinando parti di vettori diversi si ottengono sempre cammini coerenti: comunque, l'applicazione dell'algoritmo genetico con questo tipo di rappresentazione e il tradizionale operatore di *crossover* produce risultati inferiori alle altre metodologie di ricerca [21].

B.1.2 Rappresentazioni a matrici

Anche se da un lato la rappresentazione a *vettori* è quella più naturale, dall'altro non conserva l'informazione di proprietà importanti del percorso che rappresenta. In questa sezione descriviamo alcune rappresentazioni a *matrici* esistenti in letteratura.

Nella rappresentazione a *matrice di precedenze* proposta in [15], l'elemento della riga i e della colonna j vale 1 se la città i precede la città j nel percorso rappresentato, altrimenti vale 0.

Le condizioni affinchè una matrice nxn rappresenti un percorso legale per il TSP sono le seguenti:

- gli elementi della matrice con lo stesso indice di riga e lo stesso indice di colonna sono uguali ad 0 (una città ovviamente non è preceduta da se stessa);
- transitività della relazione di precedenza, cioè se una citta i precede la città j (l'elemento della matrice m_{ij} vale 1) e la città j precede la città k (m_{jk}=1), allora la città i precede la città k (m_{ik} =1);
- il numero di 1 della matrice è

$$\frac{n(n-1)}{2}$$

  in quanto l'ultima città del percorso è preceduta da tutte le precedenti che sono *n-1* ; quella immediatamente precedente da *n-2* , e cosí via.

Un'altra soluzione proposta in [24] prevede che l'elemento di riga i e colonna j vale 1 se c'è un arco che collega le città i e j, altrimenti vale 0. Il numero legale di 1 in una matrice di questo tipo è ovviamente uguale al numero di città del cammino.

B.2 Operatori Genetici

In questo paragrafo sono descritti gli operatori genetici applicati al problema del commesso viaggiatore esistenti in letteratura.

Tali operatori devono garantire sempre la generazione di percorsi validi e, per quanto riguarda il *crossover*, si differenziano in base alla rappresentazione genetica utilizzata.

B.2.1 Mutazione

L'operatore di mutazione più intuitivo applicato al TSP, è quello che cambia il collegamento tra quattro città:

si scelgono casualmente due archi (c_i,c_{i+1}) (c_j,c_{j+1}) nel percorso T, e si rimpiazzano con (c_i,c_j) e (c_{i+1},c_{j+1}), formando un nuovo percorso T'.

Questo operatore di mutazione è equivalente ad una mossa 2-opt, l'operatore basico dell'algoritmo euristico di Lin–Kernighan [31].

Un altro operatore di mutazione che modifica quattro archi del percorso è definito come segue:

Si scelgono due città casualmente, indichiamole con c_i,c_j. Si rimuovono gli archi (c_{i-1},c_i),(c_i,c_{i+1})(c_{j-1},c_j),(c_j,c_{j+1}) e si rimpiazzano con (c_i,c_{j-1}),(c_i,c_{j+1}),(c_j,c_{i-1}), (c_j,c_{i+1}).

B.2.2 Crossover

Gli operatori di *crossover* si differenziano a seconda del tipo di rappresentazione genetica che si è utilizzata.

Nella rappresentazione ad *adiacenza* possiamo distinguere i seguenti operatori di *crossover* [21]:

- *archi alternati:* costruisce un figlio dai genitori scegliendo a caso un elemento del percorso dal primo genitore, poi seleziona un appropriato elemento dal secondo genitore; l'operatore costruisce il figlio alternando gli elementi dei genitori. Inoltre, l'operatore man mano che vengono selezionati gli elementi controlla, che non si formino dei cicli prematuri. Se un nuovo arco introduce un ciclo nel percorso corrente, l'operatore seleziona un elemento che non introduce un ciclo dai rimanenti archi;

- *sottopercorsi alternati:* è una piccola variante rispetto al precedente operatore. Tale operatore costruisce il figlio selezionando alternativamente pezzi di percorso anzichè singoli elementi;

- *euristico:* costruisce un figlio partendo da una città scelta a caso nel percorso da uno dei due genitori e, tra le città che seguono questa in ciascuno dei due genitori, sceglie quella a cui corrisponde il percorso più breve. Quest'ultima città rappresenta il punto di partenza per il prosieguo della costruzione del cammino. Quest'ultimo è costruito applicando lo stesso criterio, cercando cioè tra le città che la seguono quella più vicina. Se ad un certo punto vengono introdotti dei cicli prematuri, la successiva città viene scelta a caso tra quelle che non producono cicli.

Nella rappresentazione a *percorso* distinguiamo i seguenti operatori di *crossover* :

- PMX: (proposto in [16]) costruisce i figli selezionando un sottopercorso da uno dei due genitori e cercando di preservare l'ordine con cui appaiono le città nell'altro genitore. Un sottopercorso viene selezionato scegliendo due punti a caso nella rappresentazione. Ad esempio, i due genitori

  $$p_1 = (123|4567|89)$$

  e

  $$p_1 = (452|1876|93)$$

  producono i figli nel seguente modo (le linee indicano i due punti di *crossover*). Prima, il sottopercorso tra i due punti viene scambiato producendo:

  $$o_1 = (xxx|1876|xx)$$

  e

  $$o_2 = (xxx|4567|xx).$$

Questo scambio definisce un *mapping* nel seguente modo:

$$1 <--> 4, 8 <--> 5, 7 <--> 6, 6 <--> 7.$$

Successivamente il percorso è ricostruito dai genitori originali: in o_1 la prima x dovrebbe essere 1, in quanto deve preservare l'ordine del genitore p_1, ma poichè c'è un conflitto perchè 1 è già presente nel percorso di mezzo, 1 è rimpiazzato da 4 dal mappaggio $1 <--> 4$. Allo stesso modo vengono rimpiazzate le x rimanenti. I figli generati da questo tipo di *crossover* sono:

$$o_1 = (423|1876|59)$$

e

$$o_2 = (182|4567|93).$$

- *OX*: (proposto in [13]) è molto simile al precedente operatore. Questo operatore seleziona un sottopercorso in un genitore (che non viene scambiato) e rimpiazza le altre posizioni cercando di preservare l' ordine con cui le città appaiono nell'altro genitore, omettendo le città che sono già presenti. Per esempio, i due genitori

$$p_1 = (123|4567|89)$$

e

$$p_2 = (452|1876|93)$$

producono i figli nel seguente modo (le linee indicano i due punti di *crossover*): prima il sottopercorso dei due parenti viene copiato nei figli senza fare lo scambio:

$$o_1 = (xxx|4567|xx)$$

e

$$o_2 = (xxx|1876|xx).$$

Poi iniziando dal secondo punto del genitore p_1, le città dell'altro genitore p_2 sono copiate nello stesso ordine in o_1, omettendo città già presenti; raggiunta la fine della stringa si comincia dalla prima posizione. I figli generati di questo tipo di *crossover* sono:

$$o_1 = (182|4567|93)$$

e

$$o_2 = (423|1876|59).$$

- *ER:* (proposto in [51]) è il più interessante in quanto più che le città, considera prioritari gli archi. L'operatore ER lavora costruendo una lista di città di entrambi i cammini dei genitori, in cui, per ciascuna città, sono elencate le città ad essa collegate mediante archi. La costruzione del discendente inizia selezionando una città da uno dei due genitori. Quindi viene ispezionata la lista di città collegate alla città prescelta, e la città selezionata è quella che tra queste possiede la lista degli archi più piccola. Se il numero di archi è lo stesso, allora la scelta per la città successiva viene fatta a caso.

Nella rappresentazione a matrice, proposta in [15] gli operatori di riproduzione sono denominati *intersezione* e *unione*.

L'operatore di intersezione inizia a costruire la matrice discendente (provvisoria) eseguendo l'intersezione degli elementi delle matrici dei genitori: ciascun elemento della matrice discendente vale 1 solo se entrambi i genitori hanno 1 nella stessa posizione. Questa prima fase assicura che il numero di uno non sia più grande di $\frac{n(n+1)}{2}$. La fase successiva dell'operatore di intersezione consiste nel completare il percorso, che è solo parziale. In breve, vengono selezionati alcuni elementi da uno dei due genitori che non compaiono nell'altro, e quindi vengono aggiunti alla matrice discendente.

L'operatore di unione funziona combinando sottoinsiemi di elementi dei due genitori, che sono disgiunti tra loro. Nella prima fase le città nelle matrici vengono divise in due gruppi, nel discendente vengono copiati gli elementi del primo gruppo di città dal primo genitore, e gli elementi del secondo gruppo dal secondo genitore. Infine, il cammino del discendente viene completato nello stesso modo visto per l'operatore d'intersezione.

Nella rappresentazione a matrici proposta in [24], gli operatori di riproduzione sono più simili al tradizionale operatore di *crossover*. Infatti tale operatore funziona scambiando tutti gli elementi delle due matrici genitore, dopo un certa linea (crossover ad una linea). Con questo tipo di operatore di *crossover* è necessario un algoritmo riparatore che elimina le eventuali duplicazioni di righe e colonne, quindi atto a ripristinare la legalità del cammino prodotto.

Appendice C

Definizione del problema del commesso viaggiatore

Il problema del commesso viaggiatore è definito come segue:
dato un insieme di n città e un modo di ottenere la distanza tra di esse, si trovi un cammino di lunghezza minima che passi per tutte le città ritornando alla città di partenza, passando per ogni città una sola volta.

Più formalmente il TSP si può definire nel seguente modo:
siano $C = (c_1, c_2, ..., c_n)$ le città e, date due città qualsiasi c_i c_j, sia $d(c_i, c_j)$ la distanza che le separa.

Il problema è trovare una permutazione $\pi\prime$ delle città

$$(c_{\pi'(1)}, c_{\pi'(2)}, ..., c_{\pi'(n)})$$

tale che

$$\sum_{i=1}^{n} d(c_{\pi'(i)}, c_{\pi'(i+1)}) \leq \sum_{i=1}^{n} d(c_{\pi^k(i)}, c_{\pi^k(i+1)}) \qquad \forall \pi^k \neq \pi' \qquad \text{(C.1)}$$

Dove $(n + 1) \equiv 1$.

Bibliografia

[1] E. Aarts, F. de Bront, E. Habers, and P. van Laarhoven. Parallel implementations of the statistical cooling algorithm. *Integration, the VLSI Journal*, 4:209–238, 1986.

[2] E.H.L. Aarts and P.J.M. van Laarhoven. *Simulated Annealing: Theory and Applications*. D.Reidel Publishing Company, 1987.

[3] F. Baiardi and S. Orlando. Strategies for a massively parallel implementation of simulated annealing. In *Proc. of Int. Conf. PARLE '89*, pages 273–287, June 1989.

[4] R. Baraglia and D. Laforenza. Aspetti pratici ed esperienze d'uso di una architettura in ambiente scientifico general purponse. *Atti del Congresso Annuale AICA*, 4:699–724, 1994.

[5] A.A. Bertossi. *Strutture, algoritmi, complessità*. ECIG, 1990.

[6] R. Bianchini and C.M. Brown. Parallel genetic algorithm on distribuited-memory architectures. Technical Report TR 436, Computer Sciences Department University of Rochester, 1993.

[7] H. Braun. On solving travelling salesman problems by genetic algorithms. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN , volume 496 of Lecture Notes in Computer Science*, pages 129–133. Springer-Verlag, 1991.

[8] E. Cantu-Paz. A summary of research on parallel genetic algoritms. Technical Report 95007, University of Illinois at Urbana-Champaign, Genetic Algoritms Lab. (IlliGAL), http://gal4.ge.uiuc.edu/illigal.home.html, July 1995.

[9] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli. A parallel simulated annealing algorithm for the placement of macro-cells. *IEEE Transaction on Computer-Aided Design*, CAD-6:838–847, Sept. 1987.

[10] S. Cohoon, J. Hedge, S. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. *IEEE Transaction on CAD*, 10(4):483–491, April 1991.

[11] F. Darema, S. Kirkpatrick, and V.A. Norton. Parallel algorithms fo chip placement by simulated annelaing. *IBM J.Res.Develop*, 31:391–402, May 1987.

[12] Y. Davidor. A naturally occurring niche and species phenomenon. The model and first results. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 257–263. Morgan Kaufmann Publishers, 1991.

[13] L. Davids. Applying adaptive algorithms to epistatic domains. In *Proc. of the International Joint Conference on Artificial Intelligence*, pages 162–164, 1985.

[14] M. Dorigo and V. Maniezzo. Parallel genetic algorithms: Introduction and overview of current research. In *Parallel Genetic Algorithms*, pages 5–42. IOS Press, 1993.

[15] B.R. Fox and M.B. McMahon. *Foundations of Genetic Algorithms*, pages 284–300. Morgan Kaufmann, 1991.

[16] D. Goldberg and R. Lingle. Alleles, loci, and the tsp. In *Proc. of the First International Conference on Genetic Algorithms*, pages 154–159, 1985.

[17] D.E. Goldberg. *Genetic algorithms in search, optimization and machine learning.* Addison Wesley, 1989.

[18] V. Gordon and D. Whitley. Serial and parallel genetic algorithms as function optimizers. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 177–183, 1993.

[19] M. Gorges-Schleuter. Explicit parallelism of genetic algorithms through population structures. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN*, volume 496, pages 398–406. Lecture Notes in Computer Science, 1990.

[20] M. Gorges-Schleuter. *Genetic Algoritms and Population Structure.* PhD thesis, University of Dortmund, 1991.

[21] J.J. Grefenstette, R. Gopal, and B. Rosmaita. Genetic algorithm for the tsp. In *Proc. of the First International Conference on Genetic Algorithms*, pages 160–168, 1985.

[22] P. Grosso. *Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus Model.* PhD thesis, University of Michigan, 1985.

[23] J. Holland. *Adaptation in Natural and Artificial Systems.* Univ. of Mitchigan Press, 1975.

[24] A. Homaifar and S. Guan. A new approch on the traveling salesman problem by genetic algorithm. Technical report, North Carolina A & T State University, 1991.

[25] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation. In *Workshop on statistical physics in engineering and biology*. Yorktown Heights, 1986.

[26] K.A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, 1975.

[27] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[28] J. Koza and D. Andre. Parallel genetic programming on a network of transputer. Technical Report CS-TR-95-1542, Stanford University, 1995.

[29] S.A. Kravitz and R. Rutenbar. Placement by simulated annelaing on a multiprocessor. *IEEE Transaction on Computer-Aided Design*, CAD-6:534–549, 1987.

[30] S. Lin. Computer solutions of the traveling salesman problem. *Bell Syst. Tech. Journal*, 44:2245–2269, 1965.

[31] S. Lin and B. Kerninghan. An effective heuristic for the traveling salesman problem. *Oper. Res.*, 21:498–516, 1973.

[32] S. Lin, W. Punch, and E. Goodman. Coarse-grain parallel genetic algorithms: Categorization and new approch. In *Sixth IEEE Symposium on Parallel and Distributed Processing*, 1994.

[33] F. Luccio. *La struttura degli algoritmi*. Bollati Boringhieri, 1982.

[34] B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 428–433. Morgan Kaufmann Publishers, 1989.

[35] O. Martin and S.W.Otto. Combining simulated annealing with local search heuristic. To appear on Annals of Operation Research.

[36] O. Martin, S.W.Otto, and E.W. Felten. Large step markov chain for the traveling salesman. *J.Complex Syst. 5:3:299*, 1991.

[37] N. Metropolis, A.M. Rosenbluth, and A.E. Teller. Equations of state calculations by fast comp. machines. *J. of Chem. Physics*, 21:1087–1092, 1953.

[38] Z. Michalewicz. *Genetic Algorithms + data structures = Evolution Programs*. Springer-Verlag, 1994.

[39] H. Muhlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 416–422, 1989.

[40] H. Muhlenbein. Evolution in time and space: the parallel genetic algorithm. In *Foundations of Genetic Algorithms*, pages 316–337. Morgan Kaufmann, 1991.

[41] H. Muhlenbein. Parallel genetic algorithms, population genetic and combinatorial optimization. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN*, volume 496, pages 407–417. Lecture Notes in Computer Science, 1991.

[42] H. Muhlenbein, M. Gorges-Schleuter, and O. Kramer. Evolution algorithms in combinatorial optimization. *Parallel Computing*, 7:65–88, 1988.

[43] H. Muhlenbein, M. Schomisch, and J. Born. The parallel genetic algorithm as function optimizer. *Parallel Computing*, 17:619–632, 1991.

[44] nCUBE Corporation. ncube2 processor manual. 1990.

[45] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs,NJ: Prentice Hall, 1982.

[46] R. Perego. Ottimizzazione parallela con simulated annealing e calcolo speculativo. *Rivista di Informatica, AICA*, XXV:25–38, Gennaio-Marzo 1995.

[47] C. Pettey, M. Lenze, and J. Grefenstette. A parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 155–161. L. Erlbaum Associates, 1987.

[48] M. Schwehm. Implementation of genetic algorithms on various interconnections networks. *Parallel Computing and Transputer applications*, pages 195–203, 1992.

[49] R. Tanese. Parallel genetic algorithms for a hypercube. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 177–183. L. Erlbaum Associates, 1987.

[50] R. Tanese. Distribuited genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 434–440. M. Kaufmann, 1989.

[51] D. Whitley, T. Starkweather, and D.A. Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *Proc. of the Third International Conference on Genetic Algorithms*, pages 133–140, 1989.

[52] E.E. Witte, R.D. Chamberlain, and M.A. Francklin. Parallel simulated annnealing using speculative computation. *IEEE Transaction on Parallel and Distribuited Systems*, 2:483–494, October 1991.