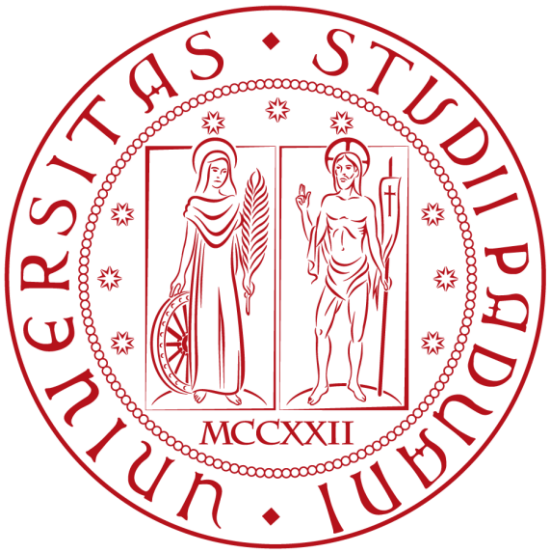

Università degli studi di Padova

Facoltà di Ingegneria Gestionale
Dipartimento di tecnica e gestione dei sistemi industriali
Tesi di Laurea di Primo Livello

TECNICHE EURISTICHE PER LA RISOLUZIONE DEL PROBLEMA DEL COMMESSO VIAGGIATORE

Relatore: Ch.mo Prof. Romanin Jacur Giorgio

Laureando: Adami Matteo
Matricola: 575421

ANNO ACCADEMICO 2010/2011

Indice

Sommario

Con tale lavoro si intende fornire una panoramica generale sui metodi euristici più comunemente utilizzati per risolvere quello che in bibliografia è conosciuto come il *problema del commesso viaggiatore*. Dopo una breve introduzione dove si esplicitano alcuni concetti base, importanti per la trattazione successiva, riguardanti i problemi di logistica distributiva la tesi procede con un capitolo iniziale nel quale vengono enunciate le nozioni fondamentali di teoria dei grafi, necessarie per comprendere le successive spiegazioni. Nel capitolo 2, invece, si iniziano ad circoscrivere i confini della trattazione occupandosi in particolare dei problemi di ottimizzazione, con particolare attenzione alle tematiche della complessità computazionale e dei problemi di ottimizzazione combinatoria. Il capitolo 3, che consiste nel corpo centrale dell'elaborato, infine, si occupa nello specifico del problema del commesso viaggiatore discutendo della sua formulazione matematica e descrivendo le principali tecniche euristiche e meta-euristiche di risoluzione dello stesso.

Introduzione

I Problemi di *logistica distributiva*, detti anche problemi di *routing*, riguardano l'organizzazione di sistemi di distribuzione di beni e servizi. Esempi di problemi di questo genere sono la movimentazione di pezzi o semilavorati tra reparti di produzione, la raccolta e distribuzione di materiali, la distribuzione di merci da centri di produzione a centri di distribuzione (es:raccolta rifiuti, distribuzione della posta). Dal punto di vista astratto un problema di routing si può schematizzare attraverso un insieme di clienti, aventi una certa localizzazione territoriale, a ciascuno dei quali deve essere erogato una certa tipologia di servizio, riguardante un certo bene, ovvero:

-consegna

-prelievo

-consegna/prelievo (contestuale necessita di prelievo in una località di origine e consegna in una di destinazione)

Tali tipologie di servizio possono appartenere poi alla categoria dei problemi con *finestre temporali* (*time windows)* nel caso in cui esista un intervallo di tempo entro il quale il servizio stesso deve essere effettuato.

Se non sussistono vincoli temporali la questione si riduce a definire la successione delle località che ogni mezzo utilizzato nel servizio deve effettuare (problema di *routing puro*), mentre in caso opposto oltre alla definizione dei percorsi si deve considerare la loro tempificazione (*problema di routing e scheduling).*

Se inoltre i veicoli utilizzati nel trasporto presentano una certa capacità massima di trasporto allora si parla di problemi con *vincoli di capacità.*

Il problema può presentare 2 ulteriori caratteristiche:

• staticità, se tutte le informazioni necessarie per soddisfare le richieste di servizi che vengono commissionati sono disponibili prima della scelta del come soddisfare tali richieste

• dinamicità, se le specifiche del problema possono mutare a servizio già avviato (*servizi a chiamata*).

Nel caso in cui i dati a disposizione per pianificare le soluzioni del problema siano forniti in termini di statistiche si ha a che fare con problemi di routing *stocastico*; nel caso invece che i dati siano noti e certi si parla di problema di routing *deterministico.*

Per descrivere queste tipologie di problemi di logistica distributiva risulta di semplice comprensione e di comprovata efficacia utilizzare come strumento di rappresentazione

il grafo pesato in modo da poter modellizzare una generica località geografica con un nodo del grafo e rappresentare i possibili collegamenti tra le località stesse con gli archi del grafo i cui pesi possono, a seconda delle specifiche del problema, indicare la lunghezza del tratto di collegamento, il tempo necessario a percorrerlo o il costo per percorrerlo.
L'obbiettivo che generalmente ci si prefigge di perseguire nello studio di tali problemi è la minimizzazione dei costi necessari ad erogare i servizi richiesti; si può distinguere a tal proposito, sulla base della loro natura, i costi che vanno considerati nel problema in:

• *costi fissi*, dovuti essenzialmente agli investimenti
• *costi variabili*, legati strettamente all'effettuazione del servizio.

In questi ultimi, ossia nei costi variabili, si possono individuare ulteriori 2 categorie di distinzione ossia i *costi di utilizzazione dei veicoli* (es: personale, assicurazione, ecc…) e *costi di routing* ossia quelli direttamente proporzionali alla lunghezza del tragitto da percorrere (es: benzina).
Il largo numero di applicazioni al mondo reale ha ampliamente dimostrato che l'uso di procedure computerizzate produce sostanziali risparmi (dal 5 al 20%) nei costi globali di trasporto rispetto ai casi in cui le decisioni sono prese quasi esclusivamente secondo l'esperienza e il buon senso. Questo diventa ancora più importante se si considera che il processo di trasporto, coinvolgendo tutte le fasi della produzione, rappresenta una parte rilevante (dall'11% al 13%) del costo totale della produzione; anche un solo piccolo miglioramento di queste voci può equivalere ad un buon risultato e una trattazione algoritmica rigorosa diventa estremamente utile.
Se i servizi si intendono localizzati nei nodi i problemi di routing descritti per mezzo di grafo si dicono a copertura dei nodi (*node covering)* e l'obbiettivo è quello di ricercare una gamma di percorsi che visitano i nodi identificativi dei servizi. Ad ogni nodo è possibile associare un peso che rappresenta la quantità di domanda del servizio da realizzare.
Nei problemi a copertura di archi (*arc covering*), invece, i servizi si ipotizzano localizzati lungo gli archi del grafo, pertanto ci si prefigge di calcolare dei percorsi che contengano gli archi nei quali sono presenti i servizi (es: consegna della posta).
Nella trattazione che segue ci si soffermerà esclusivamente sui problemi di tipo node covering di cui le principali formulazioni affrontate in letteratura sono il *problema del commesso viaggiatore* e il *problema del Vehicle Routing*.
Il *problema del commesso viaggiatore* (*Travelling Salesman Problem – TSP*) consiste nella determinazione di un circuito a costo minimo che passi per ciascun nodo almeno

una volta. Nel caso di grafo completo in cui sia valida la disuguaglianza triangolare la soluzione coincide con un *circuito hamiltoniano* ovvero un circuito che passi per ciascun nodo una sola volta. Se il costo dello spostamento non dipende dalla direzione secondo la quale lo si effettua si parla di *TSP simmetrico*, in caso opposto di *TSP asimmetrico.*

Il *problema del Vehicle Routing* (*Vehicle Routing Problem – VRP*) ricerca i percorsi da assegnare a ciascuno dei veicoli disponibili presso un deposito che visitino tutti i nodi del grafo minimizzando la distanza totale percorsa. Se il problema è sottoposto a vincoli di capacità e a ciascun nodo si associa una domanda non negativa si parla di *Vehicle Routing con vincoli di capacità* (*Capacited Vehicle Routing Problem – CVRP*) e l'obbiettivo è quello di ricercare i percorsi di costo totale minimo tali che la somma delle domande associate ai nodi di ogni percorso non ecceda la capacità dei veicoli. In assenza di vincoli di capacità e se il numero dei veicoli è stabilito a priori il *VRP* viene altresì detto problema del commesso viaggiatore multiplo (Multiple Travelling Salesman Problem - *MTSP*) dal momento che si può considerare come una estensione del *TSP* caratterizzata dal fatto che vi sono più veicoli i cui percorsi hanno tutti inizio e fine da uno stesso nodo che è quindi comune a tutti gli itinerari trovati. Nel seguito del lavoro si porrà l'accento solo sulla trattazione del TSP asimmetrico.

Capitolo 1

Nozioni base di teoria dei grafi

1. Concetti base

Si definisce grafo una figura costituita da punti, detti vertici o nodi, e da linee che li uniscono, detti lati o spigoli o archi. Più formalmente, dati un insieme V di nodi e un insieme E di coppie di elementi di V costituenti collegamenti tra vertici, un grafo G è una coppia ordinata G = (V, E). In termini grafici i grafi vengono spesso rappresentati con immagini con cerchi e linee, i cerchi identificano i vertici mentre le linee congiungenti rappresentano i lati o gli archi a seconda della tipologia di grafo. Esistono molteplici catalogazioni dei grafi a seconda della caratteristica su cui si concentra l'attenzione. Al fine del problema del commesso viaggiatore ci concentreremo solo sulla classificazione dei grafi orientati e non orientati.

1.1.Grafi non orientati

Un *grafo non orientato* G = (V, E) si distingue dal fatto che E è una famiglia di coppie non ordinate di elementi di V. Gli elementi di V sono chiamati *vertici* di G mentre gli elementi di E sono chiamati *spigoli* di G. In alcuni casi può essere specificata una funzione c: $E \to \Re$ o una funzione w: $V \to \Re$, in tal caso G si dice rispettivamente *pesato sugli spigoli* o *pesato sui vertici.* Tipicamente V è definito come V = {1,…, n} mentre gli spigoli hanno la forma {*i, j*} con *i, j* $\in$ V. Dato che il grafo è non orientato la connessione [*i, j*] ha lo stesso significato della connessione [*j, i*]. In figura 1 è rappresentato un grafo non orientato G = (V, E) con vertici V = {1,2,3,4,5,6} e spigoli [1, 1], [1, 2], [1, 3], [1, 4], [2, 4], [3, 4], [3, 4], [4, 5]. Si noti che il lato [3, 4] compare due volte costituendo uno *spigolo multiplo* o *spigolo parallelo*. Grafi senza spigoli multipli si dicono *grafi semplici*. In questa sede non saranno considerati i *multigrafi*, ossia i grafi non semplici. Spigoli di un grafo nella forma [*i, i*] si dicono *cappi*, un esempio è dato dal collegamento [1, 1] in figura 1. In generale si dirà che il lato [*i, j*] *collega* o è *incidente* nei vertici *i* e *j*, in questo caso *i* e *j* si diranno *adiacenti*. Due spigoli si dicono *adiacenti* se hanno un vertice in comune. Si definisce grado di un vertice *v* il numero degli spigoli che vi incidono e si indica con gr(*v*). Se un vertice ha grado nullo, cioè gr(*v*)=0, il vertice si dice *isolato* (vertice 6 in figura), se il grado è pari a uno, cioè gr(*v*)=1 il vertice si dice *pendente* (vertice 5 in figura).

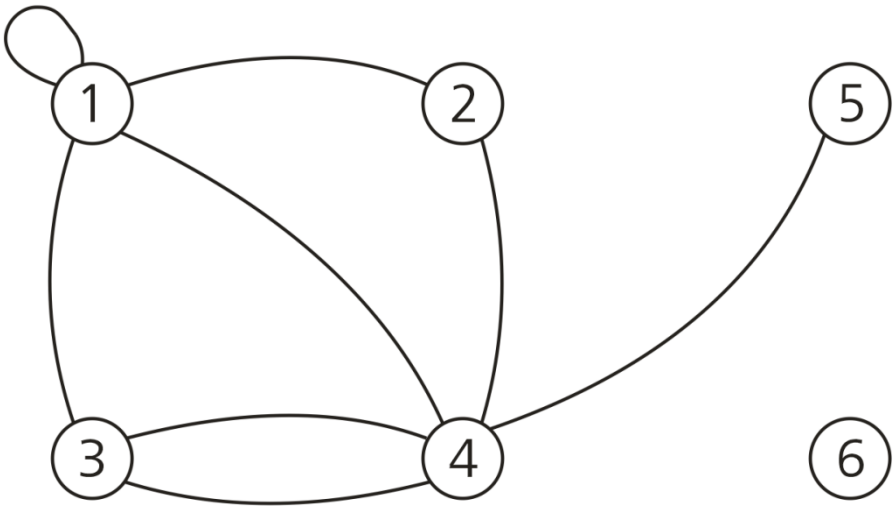

Figura 1 - Grafo non orientato.

In generale vale sempre il seguente Teorema: la somma dei gradi di tutti i vertici di un grafo G = (V,E) è un numero pari, uguale al doppio del numero degli spigoli del grafo; in formula:

$$\sum_{v \in V} gr(v) = 2 \mid E \mid$$

Corollario**:** La somma dei gradi dei vertici di grado dispari è pari.

Una *catena* di G è costituita da una sequenza di spigoli adiacenti e_1, e_2,…, e_n tali che ciascuno spigolo e_i ha un vertice estremo in comune con e_{i-1} e l'altro in comune con e_{i+1} (i=2, 3, …, n-1). Ad esempio ([1, 2], [2, 4], [4, 5]) di Figura 1. Una catena è *semplice* se gli spigoli che la compongono sono tutti distinti. Una catena è *chiusa* se il vertice estremo di e_n che non è in comune con e_{n-1} coincide con il vertice estremo di e_1 che non è in comune con e_2. Una catena semplice e chiusa si chiama *ciclo* (ad esempio ([1, 2], [2, 4], [4, 3], [3, 1]). La catena si dice *elementare* se i vertici che la compongono sono tutti distinti, ad esempio la catena (1,2,4,3) è elementare, mentre la catena (2,4,3,1,4) è semplice ma non elementare. Un vertice *v* si dice *connesso* ad un altro vertice *w* se esiste una catena che li collega; si noti che la proprietà è simmetrica. Per definizione ogni vertice è connesso a se stesso.

Un grafo si dice *connesso* se per ogni coppia di vertici esiste a catena che li congiunge (il grafo in figura 1 non è connesso perché ha il vertice 6 isolato). Un grafo G si dice

completo se, presi comunque due vertici distinti v_1 e $v_2 \in$ V esiste uno spigolo che ha v_1 e v_2 come estremi. Il grafo di figura 1 non è completo. Una catena o ciclo elementare si dice *euleriano* se usa una ed una sola volta tutti i lati del grafo. Un grafo connesso si dice *euleriano* se esiste un suo ciclo euleriano. Un famoso teorema di Eulero dice che G è euleriano se e solo se è connesso e ogni vertice $v \in$ V ha un grado gr(v) pari. Una catena (o ciclo) semplice si dice *hamiltoniano* se visita una ed una sola volta tutti vertici del grafo. G è un *grafo hamiltoniano* se contiene un ciclo hamiltoniano. Non sono note condizioni "semplici" per verificare se un dato grafo G è hamiltoniano. Ovvia condizione necessaria (ma non sufficiente) è che G deve essere connesso con gr(v) ≥ 2 $\forall v \in$ V.

1.2.Grafi orientati

Un grafo orientato G = (N, A) si caratterizza dal fatto che A è una famiglia di coppie ordinate di elementi di N identificanti collegamenti tra questi. Gli elementi di N sono chiamati nodi di G mentre gli elementi di A sono chiamati archi di G. Anche in questo caso il grafo può essere *pesato sugli archi* con una funzione c: A $\rightarrow \Re$ o sui nodi con una funzione w: N $\rightarrow \Re$. Gli archi hanno quindi forma (*i, j)* con *i, j* $\in$ N e l'ordine tra gli elementi è importante ed in generale (*i, j*) $\neq$ (*j, i*). Dal collegamento (*i, j*) si identifica il vertice *i* come *coda* ed il vertice *j* come *testa* dell'arco che *esce* da *i* ed *entra* in *j*. Un esempio di grafo orientato è mostrato in figura 2 in cui N = {1,2,3,4,5} ed A = {(1, 2), (1, 3), (1, 4), (2, 4), (3, 1), (4, 3), (4, 5)}.

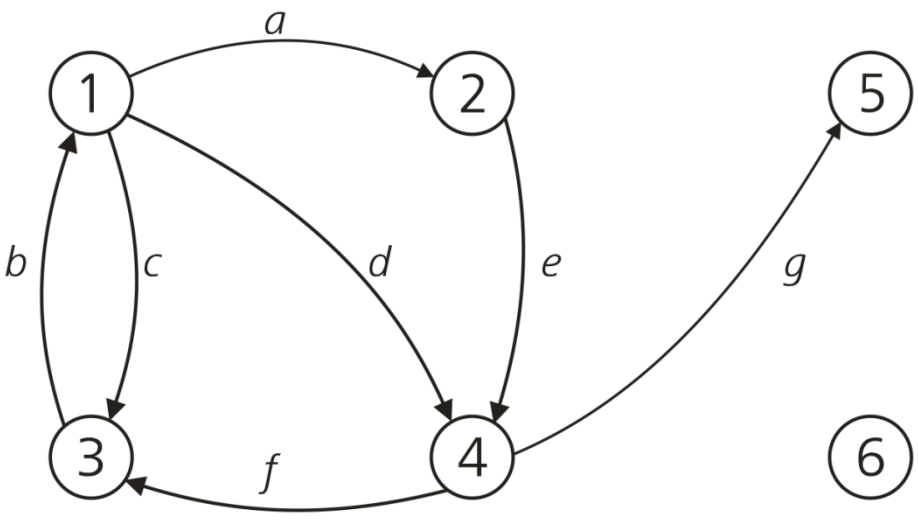

Figura 2 - Grafo orientato.

Molte delle definizioni dei grafi non orientati sono estendibili ai grafi orientati: si definiscono quindi grafi orientati semplici (cioè senza archi multipli), di loop, di cammini orientati (al posto delle catene) e di cicli orientati chiamati anche *circuiti*. In particolare un *cammino* è una sequenza di archi consecutivi a_1, a_2,…, a_n tale che ciascun arco a_i è consecutivo ad a_{i-1} (i=2, 3,…, n). Si introduce il concetto di *raggiungibilità* che è l'estensione per grafi orientati del concetto di connessione. Si noti che la proprietà non è simmetrica: in figura 2 si può notare che il nodo 5 non può raggiungere 4. Per grafi orientati si aggiunge inoltre il concetto di forte connessione: un grafo orientato è *fortemente connesso* se per ogni coppia di nodi (v_1, v_2) esiste un cammino orientato che li unisce, pertanto ogni nodo è raggiungibile da ogni altro nodo. Un grafo orientato fortemente connesso si dice *euleriano* se esiste un circuito orientato che usa una ed una sola volta tutti gli archi del grafo. Un grafo orientato è *hamiltoniano* se esiste un circuito semplice che passa una ed una sola volta per tutti i vertici del grafo. Come per il caso non orientato non sono note caratterizzazioni "semplici" dei grafi hamiltoniani.

1.3.Rappresentazione di grafi

Per rappresentare grafi sono possibili molteplici formalizzazioni matematiche che evidenzino particolari caratteristiche del grafo. Successivamente si fornirà descrizione della rappresentazione mediante matrice di adiacenza vertici-vertici.

1.3.1.Matrice di adiacenza vertici-vertici

Dato un grafo non orientato G = (V, E) si definisce come matrice di adiacenza vertici-vertici, la matrice quadrata D = [d_{ij}] di ordine n = |V| ed i cui elementi d_{ij} valgono:

$$d_{ij} = \begin{cases} +1 & \text{se } i \text{ e } j \text{ sono vertici estremi di un medesimo arco} \\ 0 & \text{altrimenti} \end{cases}$$

La matrice di adiacenza del grafo in figura 2 è:

$$D = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Si noti che il vertice isolato 6 ha la riga e la colonna corrispondente sempre nulla, la memorizzazione di tale struttura richiede, infatti, sempre uno spazio pari a $|N|^2$ indipendentemente dal numero di spigoli del grafo. Spesso può essere utile disporre della matrice di adiacenza pesata sugli spigoli Q', in questo caso gli elementi della matrice saranno nella forma:

$$q_{ij}' = q_{ij} \cdot c(a_{ij})$$

dove $c(a_{ij})$ rappresenta il *peso* o il *costo* dello spigolo tra v_i e v_j.

Capitolo 2

Problemi di ottimizzazione

2.Definizione

Molti dei processi decisionali affrontati per la gestione di un'impresa o di un'organizzazione complessa possono essere ricondotti al seguente schema logico: a fronte del problema considerato, il decisore individua un insieme di decisioni attuabili, e stabilisce un criterio di valutazione e confronto tra scelte alternative, quale ad esempio il costo o il guadagno monetario determinati dalla decisione. A questo punto, il decisore è chiamato ad individuare la decisione "migliore" rispetto al criterio di valutazione introdotto, ovvero la scelta che rende minimo il costo o massimo il guadagno.
Sul piano della rappresentazione matematica, il paradigma concettuale riassunto determina una categoria molto estesa di modelli quantitativi: i modelli di ottimizzazione.
La forma generale che un modello di ottimizzazione deve possedere presuppone che dato un insieme X, che costituisce l'insieme delle soluzioni ammissibili, e una funzione reale *c(x)* definita per ogni elemento dell'insieme X, chiamata *funzione obiettivo*, si richiede di determinare l'elemento di X che minimizza la funzione *c(x)*. In simboli, occorre risolvere il problema di ottimizzazione

$$\begin{aligned} &\min c(x) \\ &x \in X \end{aligned} \qquad (P)$$

L'insieme X delle soluzioni ammissibili viene anche detto *insieme ammissibile* oppure *regione ammissibile*. Gli elementi $x^* \in X$ che risolvono il problema *P*, ovvero che minimizzano la funzione *c(x)*, vengono detti *soluzioni ottimali*. Il corrispondente valore dell'obiettivo $c^*=c(x^*)$ viene detto *ottimo*. In generale, non è detto che esista sempre almeno una soluzione ottimale, e, ove esista, non è detto che sia unica. In particolare, se l'insieme ammissibile X è vuoto si dice che il problema *P* è *inammissibile*. Se invece accade che per ogni valore $k > 0$ esiste un punto $x \in X$ tale per cui $c(x) < -k$, allora diremo che il problema P risulta essere *inferiormente illimitato*, o più semplicemente *illimitato*.
Interpretando il significato dei concetti introdotti, in riferimento ad un processo decisionale, si può dire che le soluzioni ammissibili dell'insieme X rappresentano le decisioni che il decisore ritiene accettabili nello specifico contesto applicativo in cui è chiamato

ad operare. La funzione obiettivo *c(x)* costituisce il criterio di prestazione sulla base del quale egli intende confrontare le decisioni ammissibili, per individuare la decisione migliore: se il criterio di prestazione esprime un costo economico, verrà considerata ottimale una decisione che renda minima la funzione obiettivo.
Il problema *P* richiede di minimizzare una funzione obiettivo, che può pertanto essere interpretata come un costo economico, una penalità, o comunque una caratteristica "indesiderata" per il decisore, responsabile della formulazione del modello. In altri casi, il decisore si trova a valutare le proprie decisioni attraverso una funzione obiettivo che rappresenta un guadagno monetario, un premio, o comunque una caratteristica "desiderata", che deve essere massimizzata anziché minimizzata. In questi casi, il generico problema P prende la seguente forma, che si può a tutti gli effetti equivalente alla precedente

$$\max c(x)$$
$$x \in X \qquad (P')$$

Il paradigma di modello di ottimizzazione, espresso simbolicamente dalla formulazione del problema *P*, presuppone che il processo decisionale abbia una struttura logica di razionalità assoluta: il decisore traduce le proprie preferenze nella funzione obiettivo, e seleziona la decisione che realizza la minima oppure la massima prestazione.
Sul piano più generale, un modello di ottimizzazione offre vantaggi di grande rilievo:

a) La formulazione stessa del modello costringe ad uno sforzo di razionalizzazione, e consente al decisore di acquisire una maggiore comprensione del problema decisionale che sta affrontando. La descrizione verbale di una regola o di un criterio di preferenza è sempre più ambigua della sua trasposizione nella simbologia matematica.

b) Un modello di ottimizzazione individua una soluzione ottimale, ma non costringe il decisore ad adottarla in pratica. Con ciò vogliamo dire che il decisore è libero di utilizzare l'informazione ricevuta, magari modificando parzialmente la soluzione proposta dal modello, per adattarla ad altri criteri di preferenza non espressi dalla funzione obiettivo. Da questo punto di vista, la conoscenza della soluzione ottimale rappresenta una soglia di riferimento nei confronti della quale giudicare la qualità di ogni altra soluzione che si voglia adottare in pratica.

c) Le tecniche risolutive per i modelli di ottimizzazione consentono di individuare una soluzione ottimale per problemi applicativi di grandi dimensioni, per i quali una corrispondente ricerca "manuale", basata soltanto sul buon senso e sull'esperienza del decisore, fornirebbe quasi certamente soluzioni di qualità inferiore.

La forma generale del problema di ottimizzazione descritta da *P* e *P* ' non richiede alcuna assunzione particolare circa la natura e le proprietà dell'insieme X e della funzione obiettivo *c(x)*. Specifiche ipotesi relative a X e *c(x)* conducono a diverse classi di problemi di ottimizzazione, che costituiscono casi particolari di *P*. Ad esempio i problemi di programmazione lineare si ricavano da *P* nell'ipotesi che l'insieme X sia un particolare tipo di sottoinsieme dello spazio n-dimensionale $\Re^n$ e che *c(x)* sia una funzione lineare.
Pertanto, la classe generale dei problemi di ottimizzazione origina una ricca tassonomia di sottoclassi, caratterizzate da differenti assunzioni circa l'insieme X e la funzione *c(x)* .Queste sottoclassi si suddividono a loro volta, dando luogo ad una gerarchia di modelli, che si distinguono per gli ambiti di applicazione, per le metodologie risolutive, per la difficoltà presentata.

2.1.Complessità Computazionale

Una volta che un problema *P* sia stato formulato, un *algoritmo* che risolve *P* può essere definito come una sequenza finita di istruzioni che, applicata ad una qualsiasi istanza *p* di *P*, si arresta dopo un numero finito di passi (ovvero di computazioni elementari), fornendo una soluzione di *p* oppure indicando che *p* non ha soluzioni realizzabili.
Per poter studiare gli algoritmi dal punto di vista della loro efficienza, o complessità computazionale, è necessario definire un modello computazionale; classici modelli computazionali sono la Macchina di Turing e la Macchina a Registri (MR).

2.1.1.Misure di Complessità

Dato un problema *P*, una sua istanza *p*, ed un algoritmo *A* che risolve *P*, indichiamo con *costo* (o *complessità*) di *A* applicato a *p* una misura delle risorse utilizzate dalle computazioni che *A* esegue su una macchina MR (presumibilmente in grado di calcolare qualsiasi funzione effettivamente computabile con procedimenti algoritmici) per determinare la soluzione di *p*.

Le risorse, in principio, sono di due tipi: *memoria occupata* e *tempo di calcolo*. Molto spesso la risorsa più critica è il tempo di calcolo (tempo di esecuzione dell'algoritmo) e quindi si usa soprattutto questa come misura della complessità degli algoritmi. Nell'ipotesi che tutte le operazioni elementari abbiano la stessa durata, il tempo di calcolo può essere espresso come numero di operazioni elementari effettuate dall'algoritmo.
Dato un algoritmo, è opportuno disporre di una misura di complessità che consenta una valutazione sintetica della sua bontà ed eventualmente un suo agevole confronto con algoritmi alternativi. Conoscere la complessità di *A* per ognuna delle istanze del problema *P* non è possibile (l'insieme delle istanze di un problema è normalmente infinito), né sarebbe di utilità pratica.
Si cerca allora di esprimere la complessità come una funzione *g(n)* della *dimensione n* dell'istanza cui viene applicato l'algoritmo, definita come una misura del numero di bit necessari per rappresentare, con una codifica ragionevolmente compatta, i dati che definiscono l'istanza, cioè una misura della lunghezza del suo input. Dato che per ogni dimensione si hanno in generale molte istanze di quella dimensione, si sceglie *g(n)* come il costo necessario per risolvere la più difficile tra le istanze di dimensione *n*; si parla allora di complessità nel caso peggiore.
A questo punto la funzione $g(n)$ risulta definita in modo sufficientemente rigoroso, continuando però ad essere di difficile uso come misura della complessità, dato che risulta difficile, se non praticamente impossibile, la valutazione di *g(n)* per ogni dato valore di *n*. Questo problema si risolve sostituendo alla funzione *g(n)* il suo ordine di grandezza; si parla allora di *complessità asintotica*.
Date due funzioni *f(x)* e *g(x),* diremo che:

1. *g(x)* è *O(f(x))* se esistono due costanti c_1 e c_2 per cui $g(x) \leq c_1 * f(x) + c_2 \ \forall x$;

2. $g(x)$ è Ω*(f(x))* se $f(x)$ è $O(g(x))$;

3. $g(x)$ è $\alpha(f(x))$ se $g(x)$ è allo stesso tempo $O(f(x))$ e Ω(($f(x)$).

Sia $g(x)$ il numero di operazioni elementari che vengono effettuate dall'algoritmo *A* applicato alla più difficile istanza, tra tutte quelle che hanno lunghezza di input *x*, di un dato problema *P*. Diremo allora che la complessità di *A* è un $O(f(x))$ se $g(x)$ è un $O(f(x))$, è un Ω(($f(x)$) se $g(x)$ è un Ω(($g(x)$) ed un α ($f(x)$) se $g(x)$ è un α ($f(x)$).
Un algoritmo è detto avere una *complessità polinomiale* se il tempo di esecuzione è dell'ordine $O(x^k)$, dove *k* è una costante indipendente dalla lunghezza di input *x*. Se la funzione di complessità di tempo non può essere limitata da un polinomio, l'algoritmo è

detto avere *complessità esponenziale*. Se l'espressione b^l, dove l è una costante ed b è il valore di input più grande, è parte della funzione che limita, cioè se il tempo di running è dell'ordine di $O(x^k * b^l)$, allora l'algoritmo è detto avere *complessità pseudo-polinomiale*.

2.1.2.Classi P e NP

Chiameremo *problemi trattabili* quelli per cui esistono algoritmi la cui complessità sia un $O(p(x))$, con $p(x)$ un polinomio in x, e *problemi intrattabili* quelli per cui un tale algoritmo non esiste.
Per poter effettuare una più rigorosa classificazione dei diversi problemi, conviene far riferimento a problemi in *forma decisionale*. Una prima importante classe di problemi è la classe *NP*, costituita da tutti i problemi decisionali il cui problema di certificato associato può essere risolto in tempo polinomiale. In altri termini, i problemi in *NP* sono quelli per cui è possibile verificare efficientemente una risposta positiva, perché è possibile decidere in tempo polinomiale se una soluzione x è realizzabile per il problema.
Equivalentemente si può definire *NP* come la classe di tutti i problemi decisionali risolubili in tempo polinomiale da una macchina MR non-deterministica (*NP* sta, infatti, per *Polinomiale nel calcolo Non-deterministico*). Una MR non-deterministica è il modello di calcolo (astratto) in cui una MR, qualora si trovi ad affrontare un'operazione di salto condizionale, può eseguire contemporaneamente entrambi i rami dell'operazione, e questo ricorsivamente per un qualsiasi numero di operazioni; si tratta cioè di un computer capace di eseguire un numero illimitato (ma finito) di calcoli in parallelo.
In altre parole, i problemi in *NP* sono quelli per cui esiste una computazione di lunghezza polinomiale che può portare a costruire una soluzione realizzabile, se esiste, ma questa computazione può essere nascosta entro un insieme esponenziale di computazioni analoghe tra le quali, in generale, non si sa come discriminare.
Un sottoinsieme della classe *NP* è la classe *P*, costituita da tutti i problemi risolubili in tempo polinomiale, ossia contenente tutti quei problemi decisionali per i quali esistono algoritmi di complessità polinomiale che li risolvono. Per questo motivo i problemi in *P* sono anche detti *problemi polinomiali*.
Chiaramente $P \subseteq NP$, ma una domanda particolarmente importante è se esistano problemi in *NP* che non appartengono anche a *P*, cioè se sia $P \neq NP$.
A questa domanda non si è in grado di rispondere, anche se si ritiene fortemente probabile che la risposta sia positiva.

2.1.3.Problemi NP completi e NP ardui

Molti problemi, anche se apparentemente notevolmente diversi, possono tuttavia essere ricondotti l'uno all'altro. Dati due problemi decisionali *P* e *Q* e supponendo l'esistenza di un algoritmo *AQ* che risolve *Q* in tempo costante (indipendente dalla lunghezza dell' input), diremo che *P* si riduce in tempo polinomiale a *Q*, e scriveremo $P \alpha Q$, se esiste un algoritmo che risolve *P* in tempo polinomiale utilizzando come sottoprogramma *AQ*. In tal caso si parla di riduzione polinomiale di *P* a *Q*.
È facile verificare che la relazione α ha le seguenti proprietà:

1. è riflessiva: $A \alpha A$;

2. è transitiva: $A \alpha B$ e $B \alpha C \rightarrow A \alpha C$;

3. $A \alpha B$ e $A \notin P \rightarrow B \notin P$;

4. $A \alpha B$ e $B \in P \rightarrow A \in P$.

Possiamo ora definire la classe dei problemi *NP completi*: un problema *A* è detto *NP* completo se $A \in NP$ e se per ogni $B \in NP$ si ha che $B \alpha A$.
La classe dei problemi *NP* completi costituisce un sottoinsieme della classe *NP* di particolare importanza. Un fondamentale teorema (dovuto a Cook, 1971) garantisce che tale classe non è vuota.
I problemi *NP* completi hanno l'importante proprietà che se esiste per uno di essi un algoritmo polinomiale, allora (per transitività) necessariamente tutti i problemi in *NP* sono risolubili in tempo polinomiale, e quindi $P = NP$.
In un certo senso, tali problemi sono i più difficili tra i problemi in *NP*. Un problema si dice *NP* completo in senso forte se per esso non esiste alcun algoritmo pseudo-polinomiale a meno che $P = NP$.
Un problema che abbia come problema di decisione un problema *NP* completo si dice problema *NP arduo* ed ha la proprietà di essere almeno tanto difficile quanto i problemi *NP* completi (a meno di una funzione moltiplicativa polinomiale). Si noti che un problema *NP* arduo può anche non appartenere alla classe *NP*. Come definito per i problemi *NP* completi, allo stesso modo un problema si dice *NP* arduo in senso forte se per esso non esiste alcun algoritmo pseudo-polinomiale a meno che $P = NP$.

Nonostante tutti gli sforzi dei ricercatori, non è stato possibile fino ad ora determinare per nessun problema *NP* arduo un algoritmo polinomiale, il che avrebbe fornito algoritmi polinomiali per tutti i problemi della classe. Questo fa oggi ritenere che non esistano algoritmi polinomiali per i problemi *NP* completi, ossia, come già detto, che sia $P \neq NP$.
Allo stato delle conoscenze attuali, quindi, tutti i problemi *NP* ardui sono presumibilmente intrattabili. Naturalmente, ciò non significa che non sia in molti casi possibile costruire algoritmi in grado di risolvere efficientemente istanze di problemi *NP* ardui di dimensione significativa (quella richiesta dalle applicazioni reali).

2.2.Problemi di ottimizzazione combinatoria

L'ottimizzazione combinatoria si occupa di oggetti di tipo discreto (grafi, insiemi) e studia problemi che consistono nel disporre, raggruppare, ordinare questi oggetti in modo ottimo. Quindi tali problemi hanno sempre un numero finito di soluzioni, il che farebbe presupporre una certa facilità nel trovare la soluzione ottima (quantomeno confrontando le soluzioni una per una) ma questo non è per nulla vero: nella maggior parte dei casi il numero delle soluzioni è, infatti, molto elevato e sorge quindi la necessità di progettare algoritmi adeguati alle varie tipologie di problemi valutandone poi l'efficienza.
All'interno dei problemi di ottimizzazione, in base alla struttura dell'insieme realizzabile, si possono distinguere due importanti classi di problemi.
Nel caso in cui un problema di ottimizzazione *P* sia caratterizzato dal fatto che in ogni sua istanza la regione realizzabile *F* contiene un numero finito di punti (e quindi la soluzione ottima può essere trovata confrontando un numero finito di soluzioni), si parla di problema di *Ottimizzazione Combinatoria* (o *Ottimizzazione Discreta*); si parla invece di problema di *Ottimizzazione Continua* se la regione realizzabile può contenere un'infinità non numerabile di punti. Mentre nei modelli di Ottimizzazione Combinatoria le variabili sono vincolate ad essere numeri interi, nei modelli di Ottimizzazione Continua le variabili possono assumere tutti i valori reali.
Tali classi di modelli rientrano nella categoria più generale della Programmazione Matematica, disciplina che svolge un ruolo di fondamentale importanza all'interno della Ricerca Operativa e che ha per oggetto lo studio di problemi in cui si vuole minimizzare o massimizzare una funzione reale definita su $\Re^n$ le cui variabili sono vincolate ad appartenere ad un insieme prefissato, vale a dire lo studio di problemi di ottimizzazione.

Solitamente l'insieme realizzabile F viene descritto da un numero finito di disuguaglianze del tipo $g(x) \geq b$, dove g è una funzione definita su $\Re^n$ a valori reali e $b \in \Re$. Formalmente, date m funzioni $g_i : \Re^n \rightarrow \Re$, $i = 1, \ldots, m;$ si esprime F nella forma

$F = \{ x \in \Re^n : g_i \geq b_i, i = 1, \ldots, m\}.$

Un problema di ottimizzazione può quindi essere riscritto nella forma:

$min \{ c(x) : g_i \geq b, i = 1, \ldots, m\}$

e viene chiamato *problema di Programmazione Matematica*. I problemi di Programmazione Matematica si possono classificare in base alla struttura delle funzioni che li definiscono. Più precisamente, si hanno problemi di:

• *Programmazione Lineare* se la funzione $c(x)$ e tutte le g_i sono lineari;

• *Programmazione Lineare Intera* se oltre alla condizione di linearità per la funzione obiettivo ed i vincoli, il vettore x delle variabili deve essere intero;

• *Programmazione Lineare Mista* se non tutto l'insieme delle variabili ma solo un suo sottoinsieme deve assumere valori interi.

Mentre i problemi di Programmazione Lineare appartengono alla classe *P*, la maggior parte dei problemi di Programmazione Lineare Intera, che sono problemi di Ottimizzazione Combinatoria, sono problemi *NP* ardui. Quindi, in generale, mentre i primi sono più semplici da risolvere, per i problemi di Programmazione Lineare Intera è necessario un algoritmo con complessità esponenziale.

Capitolo 3

Il problema del commesso viaggiatore

3.TSP (Travelling Salesman Problem)

Il *problema del commesso viaggiatore* è la versione non capacitiva dei problemi di node routing ed è cosi chiamato perché si presenta tipicamente ad un agente di commercio (o commesso viaggiatore appunto) che voglia visitare *n* clienti seguendo il percorso più breve e tornare, alla fine del viaggio, al punto di partenza.

Problemi matematici riconducibili al *TSP* furono trattati nell'Ottocento da Sir William Rowan Hamilton e da Thomas Penyngton Kirkman a proposito di un gioco da tavolo, ma una forma più generale e definita apparve solo negli anni Trenta a Vienna, Harvard e Princeton. In quegli anni veniva studiato soprattutto in ambiti statistico-economici e agrari.

Tuttavia il suo fascino di difficile problema combinatorio si affermò solo più tardi. Il primo vero passo in questa direzione fu mosso nel 1954 da parte di George Dantzig, Ray Fulkerson e Selmer Johnson che pubblicarono un metodo per risolvere il *TSP* su un campione di n = 49 città, per quei tempi un numero davvero notevole. Queste 49 città rappresentavano le capitali degli Stati Uniti e il costo del percorso era calcolato in base alle distanze stradali.

Nel 1962 ci fu un altro concorso per 33 città, nel 1977 fu bandito un concorso che collegasse le 120 principali città della Germania Federale e la vittoria andò a Groetschel. La tappa successiva se la aggiudicarono Padberg e Rinaldi nel 1987 completando il giro degli Stati Uniti attraverso 532 città. Nello stesso periodo Groetschel e Holland trovarono il *TSP* ottimale per il giro del mondo che passava per 666 mete importanti.

Ma un notevole passo avanti fu compiuto sempre da Padberg e Rinaldi che trovarono il percorso migliore su uno schema di 2392 punti fornito dalla Tektronics Incorporated.

Nel 1994 Applegate, Bixby, Chvàtal e Cook calcolarono un *TSP* per n = 7379 a partire da una logic array application presso i laboratori AT&T Bell. Sempre loro nel 1998 ampliarono la mappa americana inserendo 13509 città, ovvero tutte quelle con più di 500 abitanti. Pochi anni dopo fecero lo stesso esperimento in Germania, mappando 15112 cittadine.

Tuttavia il lavoro più sensazionale che Applegate, Bixby, Chvàtal, Cook e Helsgaun portarono a termine è stato la mappatura completa della Svezia. Nel 2004 essi trovarono il percorso più breve che permetteva di girare lo stato passando per tutte le sue 24978 città.

3.1.Formulazione matematica del TSP asimmetrico

Si consideri un grafo completo pesato orientato o non orientato G=(V,A) dove V={$v_1,\ldots, v_n$} è un insieme di *n* nodi e A={(v_i,v_j): $v_i, v_j \in$ V} è un insieme di *m* archi (se il grafo è orientato) o spigoli (se il grafo è non orientato). Ad ogni arco (v_i, v_j) è associato un costo c_{ij}. Se il grafo è non orientato $c_{ij} = c_{ji}$ per ogni (v_i, v_j). Per semplicità di notazione si indicheranno i nodi e gli archi attraverso i soli indici (es: nodo *i* invece che v_i e arco (i,j) invece che (v_i, v_j)).

Supporre il grafo completo non lede alla generalità del problema dal momento che dato un qualsiasi grafo è sempre possibile associare ad esso un grafo completo in cui a ciascun arco (i,j) i è attribuito il costo c_{ij} del percorso a minimo costo tra *i* e *j*. Il grafo completo costruito in questo modo soddisfa anche la disuguaglianza triangolare $c_{ij} \leq c_{is} + c_{sj}$ per ogni *i, s, j*.

Dato un grafo completo la soluzione del problema del commesso viaggiatore consiste nell'individuazione del *circuito hamiltoniano a costo minimo* ovvero di un circuito a costo minimo che passi una sola volta per tutti i nodi del grafo.

Introducendo l'insieme di variabili binarie X=(x_{ij}) tali che $x=1$ se l'arco (i,j) appartiene al circuito, $x_{ij}=0$ altrimenti, nel caso più generale di *TSP asimmetrico* ossia $c_{ij} \neq c_{ji}$ per ogni (i,j) una possibile formulazione in termini di programmazione matematica è la seguente:

$$z = \sum_{i=1,n} \sum_{j=1,n} x_{ij} c_{ij} \qquad Min! \qquad (1)$$

sottoposto a

$$\sum_{i=1,n} x_{ij} = 1 \qquad j=1,\ldots,n \quad (2)$$

$$\sum_{j=1,n} x_{ij} = 1 \qquad i=1,\ldots,n \quad (3)$$

$$X \in S \qquad (4)$$

$$x_{ij} = 0/1 \qquad i,j=1,\ldots,n \quad (5)$$

La funzione obbiettivo (1) rappresenta la minimizzazione del costo della soluzione. I vincoli (2) e (3) indicano che in ogni nodo *i* entra ed esce un solo arco (vincoli di assegnazione). I vincoli (4) rappresentano genericamente l'insieme delle relazioni che assi-

curano l'assenza di sottocircuiti nella soluzione. I soli vincoli di assegnazione, infatti, non assicurano che la soluzione sia costituita da un unico circuito.
Per questa ragione è necessario imporre ulteriori vincoli all'insieme delle variabili. Con riferimento alla (4), possibili espressioni di *S* sono:

$$S = (x_{ij} : \sum_{i \in Q} \sum_{j \notin Q} x_{ij} \geq 1) \ \forall Q \subset V \qquad (6)$$

$$S = (x_{ij} : \sum_{i,j \in R} x_{ij} \leq |R| - 1) \ \forall R \subset \{2,\ldots,n\} \quad (7)$$

I vincoli (6) esprimono la condizione che comunque preso un sottoinsieme di nodi di *Q* Deve esistere almeno un arco che colleghi un nodo di *Q* con un nodo non appartenente a *Q.*
I vincoli (7) assicurano che il numero di archi che unisce nodi di un qualsiasi sottoinsieme deve essere inferiore alla sua cardinalità assicurando, in questo modo, che all'interno del sottoinsieme non vi siano sottocircuiti passanti solo per tutti i nodi del sottoinsieme.
Dalla formulazione del problema si deduce che eliminando i vincoli (4) il *TSP* coincide con il problema dell'assegnazione che quindi rappresenta un rilassamento del *TSP;* la soluzione ottima del problema dell'assegnazione costituisce pertanto un limite inferiore per il problema del commesso viaggiatore.

3.2.Algoritmi euristici per la soluzione del TSP asimmetrico

Dato che molti problemi di Ottimizzazione Combinatoria sono "difficili" è spesso necessario sviluppare algoritmi euristici, ossia algoritmi che non garantiscono di ottenere la soluzione ottima, ma in generale sono in grado di fornire una "buona" soluzione ammissibile per il problema. Normalmente gli algoritmi euristici hanno una bassa complessità ma in alcuni casi, per problemi di grandi dimensioni e struttura complessa, può essere necessario sviluppare algoritmi euristici sofisticati e di alta complessità. Inoltre, è possibile, in generale, che un algoritmo euristico "fallisca" e non sia in grado di determinare nessuna soluzione ammissibile del problema, pur senza essere in grado di dimostrare che non ne esistono. La costruzione di algoritmi euristici efficaci richiede un'attenta analisi del problema da risolvere volta ad individuarne la struttura, ossia le caratteristiche specifiche utili, ed una buona conoscenza delle principali tecniche algoritmiche disponibili. Infatti, anche se ogni problema ha le sue caratteristiche specifiche,

esistono un certo numero di tecniche generali che possono essere applicate, in modi diversi, a moltissimi problemi, producendo classi di algoritmi di ottimizzazione ben definite.
Di seguito si analizzeranno due tra le principali tecniche algoritmiche utili per la realizzazione di algoritmi euristici per problemi di OC: gli algoritmi *greedy* e quelli di *ricerca locale*. Gli algoritmi euristici per la risoluzione del *TSP* si dividono quindi in algoritmi *costruttivi* e algoritmi *migliorativi.*
I primi sono orientati alla definizione di criteri per la costruzione graduale di un circuito.
Gli algoritmi greedy (voraci), infatti, determinano la soluzione attraverso una sequenza di decisioni “localmente ottime", senza mai tornare, modificandole, sulle decisioni prese. Questi algoritmi sono di facile implementazione e notevole efficienza computazionale ma, sia pure con alcune eccezioni di rilievo, in generale non garantiscono l'ottimalità, ed a volte neppure l'ammissibilità, della soluzione trovata.
I secondi invece, ossia gli algoritmi di ricerca locale, sono basati su un'idea estremamente semplice ed intuitiva: data una soluzione ammissibile, si esaminano le soluzioni ad essa “vicine" in cerca di una soluzione “migliore" (tipicamente, con miglior valore della funzione obiettivo); se una tale soluzione viene trovata essa diventa la “soluzione corrente" ed il procedimento viene iterato, altrimenti (ossia quando nessuna delle soluzioni “vicine" è migliore di quella corrente) l'algoritmo termina avendo determinato un ottimo locale per il problema.

Esistono moltissime famiglie di algoritmi euristici, ispirate ad approcci diversi, più o meno sofisticati. Nel seguito ci concentreremo sugli *algoritmi costruttivi*, cioè quelli nei quali la soluzione viene ottenuta a partire da zero, attraverso una sequenza di soluzioni parziali che vengono via via completate. Nel caso del *TSP*, ciò avviene aggiungendo ad ogni passo un nuovo nodo, fino a ottenere una soluzione ammissibile, cioè un ciclo che passi per tutti i nodi del grafo.

3.2.1.Nearest Neighbour

Questo algoritmo rappresenta la più classica delle procedure di costruzione per il TSP e si basa sull’idea di cercare di ottimizzare il percorso unendo ogni volta i nodi più vicini. Dato che l'obiettivo è trovare il ciclo di costo totale minimo, l'algoritmo parte da un nodo e comincia a percorrere l'arco di costo minimo che incide in quel nodo. In altre parole, raggiunge da esso il nodo più vicino del grafo. L'idea dell'algoritmo è che facendo ogni volta la scelta più economica si possa ottenere la soluzione complessiva-

mente più economica. Si tratta di un algoritmo cosiddetto greedy: seguire ogni passo nel modo migliore per arrivare al risultato globale migliore.
Il problema con questo tipo di algoritmi è l'impossibilità di prevedere gli sviluppi finali che essi comportano. L'idea è logica e per alcuni problemi semplici funziona, cioè genera veramente una soluzione ottima. In generale, però, trascura un aspetto fondamentale: ogni scelta elementare non solo contribuisce al costo complessivo (e quindi è bene fare scelte economiche), ma influisce sull'insieme delle scelte disponibili ai passi successivi. Questo significa che scelte molto economiche nei primi passi possono però rendere necessarie scelte molto costose nei passi successivi. Si parla spesso, al riguardo, della miopia degli algoritmi greedy, che sacrificano il bene futuro per un guadagno immediato. Nel caso del *TSP*, dato che il percorso non può mai tornare sui propri passi, ad ogni passo il numero di scelte disponibili cala. Dei vantaggi immediati in termini di tempo (lunghezza) comportano degli svantaggi ai passi successivi. Indicando con *S* l'insieme dei nodi inseriti nel circuito parziale e con *S'=V-S* la procedura si sviluppa secondo i seguenti passi:

1.Inizializzazione

Si sceglie un qualsiasi nodo *d* e si pone *t=d*, *S*={*d*}, *S'=V-{d}* e *z(S)=0*.

2.Aggiunta di un nuovo elemento alla soluzione parziale

Si individua il nodo *k* più vicino all'ultimo nodo *t* (quello collegato a *t* dall'arco di costo minore) inserito nel circuito e si collega *k* a *t.* Si pone $S=S \cup \{k\}$, $S'=S'-\{k\}$ e $z(S)=z(S)+c^*_{tk}$.

3.Criterio di arresto

Se $S'=\emptyset$ si connette *k* al primo nodo *d* in modo da chiudere il circuito e si arresta l'algoritmo. Altrimenti si pone *t=k* e si ritorna al passo 2.

Tale algoritmo funziona allo stesso modo nel caso di TSP simmetrico o asimmetrico. Poiché la soluzione finale dipende dal nodo iniziale scelto si può eseguire n volte l'algoritmo a partire da ogni nodo iniziale.

3.2.2.Nearest Insertion

Gli algoritmi di inserimento costruiscono gradualmente il circuito attraverso l'inserimento progressivo di nodi a partire da un sottocircuito iniziale. Quest'algoritmo somiglia al precedente, cercando nel contempo di ottimizzarlo.
L'algoritmo *Nearest Insertion* tenta di ovviare alla miopia dell'algoritmo Nearest Neighbour concedendo un grado di libertà in più. Viene cercato il nodo più vicino ad uno qualsiasi dei nodi del ciclo, anziché a quello terminale. Questo nuovo nodo viene poi aggiunto nella posizione che minimizza il percorso e non necessariamente al termine. L'algoritmo quindi aggiunge il nuovo nodo nella posizione tale da dar luogo al ciclo più economico, anziché necessariamente al termine del percorso. Inoltre ad ogni passo viene conservata una struttura di ciclo e non più di percorso.
Tuttavia, la forma della soluzione appare troppo contorta per essere davvero buona; il punto è che anche l'algoritmo Nearest Insertion, pur senza commettere errori macroscopici come il precedente, è miope, dato che tende a seguire i percorsi migliori solo per raggiungere i nodi vicini senza preoccuparsi, sino agli ultimi passi, dei nodi lontani. Questi vengono alla fine "serviti" nel modo migliore, ma si tratta del modo migliore consentito dalla forma che il ciclo ha ormai assunto nelle sue linee generali.
In definitiva la procedura nearest insertion si realizza in questo modo:

1.Inizializzazione

Si sceglie un nodo a piacere *d*, si individua il nodo *k* più vicino a *d* (quello collegato a *d* dall'arco di costo minore) e si costruisce un primo sottocircuito *S* del tipo *d-k-d*; si pone *S={d}*, *S'=V-{d}* e $z(S)=c_{dk}+c_{kd}$.

2.Fase di selezione

A partire da una soluzione parziale *S*, si individua il nodo k^* non ancora incluso in *S* che abbia la distanza minima da un nodo *i* del sottocircuito *S.*

3.Fase di inserimento

Si individua l'arco (i^*,j) di *S* che minimizza la somma $\Delta=c_{i^*k^*}+ c_{k^*j}+ - c_{i^*j}$ e si inserisce k^* in S tra i^* e j^*; si pone $z(S)= z(S)+\Delta$.

4.Criterio di arresto

Se il circuito è completo l'algoritmo si arresta, altrimenti si torna al passo 2.

3.2.3.Cheapest Insertion

Anche l'algoritmo *cheapest insertion* ovvero l'algoritmo dell'inserimento più conveniente appartiene alla classe degli algoritmi di inserimento. Essendo esso una variante dell'algoritmo nearest insertion, con riferimento alla descrizione della procedura testé fornita per quest'ultimo, possiamo dire che l'algoritmo cheapest insertion unisce la fase di selezione e quella di inserimento scegliendo direttamente la terna i^*, j^*, k^* che minimizzi l'e*xtramileage* $\Delta = c_{i^*k^*} + c_{k^*j^*} + - c_{i^*j^*}$

3.2.4.Farthest Insertion

In alcune versioni può differire il criterio di scelta del primo nodo *k* da unire al nodo iniziale *d* per formare il primo sottocircuito *d-k-d*. Ad esempio, come nel caso dell'algoritmo *farthest insertion*, invece del nodo più vicino, si può scegliere il nodo *k* più lontano da *d*. L'algoritmo farthest insertion tiene conto in partenza del fatto che tutti i nodi vanno visitati e si preoccupa anzitutto di assicurare una visita economica ai nodi più difficili, cioè a quelli più lontani. In altre parole, a ogni passo l'algoritmo trova il nodo più lontano dal ciclo corrente e ve lo inserisce nel modo migliore.
La procedura farthest insertion si può cosi descrivere:

1.Inizializzazione

Si sceglie un nodo a piacere *d*, si individua il nodo *k* più lontano a *d* (quello collegato a *d* dall'arco di costo maggiore) e si costruisce un primo sottocircuito *S* del tipo *d-k-d*; si pone $S=\{d\}$, $S'=V-\{d\}$ e $z(S)=c_{dk}+c_{kd}$.

2.Fase di selezione

A partire da una soluzione parziale *S*, si individua il nodo k^* non ancora incluso in *S* che abbia la distanza massima da un nodo *i* del sottocircuito *S.*

3.Fase di inserimento

Si individua l'arco (i^*,j) di S che minimizza la somma $\Delta=c_{i^*k^*}+ c_{k^*j}+ - c_{i^*j}$ e si inserisce k^* in S tra i^* e j^*; si pone $z(S)= z(S)+\Delta$.

4.Criterio di arresto

Se il circuito è completo l'algoritmo si arresta, altrimenti si torna al passo 2.

3.2.5.Algoritmo di saving

Si supponga di avere due circuiti non orientati aventi un nodo d in comune e siano *i* e *j* due nodi direttamente collegati a *d* appartenenti a 2 circuiti diversi; è possibile costruire un unico circuito introducendo lo spigolo *(i,j)* e cancellando gli spigoli *(i,d)* e *(d,j).*
Si può, allora, definire un *risparmio* o *saving* $s_{ij}=c_{id}+c_{dj}-c_{ij}$ che, se positivo, indica la diminuzione si costo ottenibile con la fusione dei due circuiti.
Nel caso asimmetrico si ha in generale che $s_{ij} \neq s_{ji}$; poiché i due circuiti sono orientati, l'algoritmo si sviluppa con la sola differenza che, per essere collegabili, è necessario che *i* e *j* siano rispettivamente origine e destinazione dei circuiti o viceversa.
Scelto un nodo *d* a piacere, si costruisce una prima soluzione "a stella" in cui ciascun nodo *i* è collegato a *d* attraverso una copia di spigoli o archi *(d,i), (i,d)*. Per ogni coppia di nodi *i, j* si calcolano i savings s_{ij} e si ordinano in senso decrescente. Si esamina nell'ordine la lista degli s_{ij} e si collegano i nodi *i* e *j* unificando i circuiti di appartenenza secondo l'operazione descritta in precedenza, a condizione che *i* e *j* siano estremi di circuiti diversi. L'operazione viene applicata iterativamente fin quando non si ottiene un unico circuito. Poiché la soluzione non dipende dalla scelta del nodo di riferimento *d,* si può eseguire l'algoritmo *n* volte a partire da ciascun nodo.
Possiamo quindi dire che l'algoritmo, nel caso simmetrico, si compone dei seguenti passi:

1.Inizializzazione

Si sceglie un nodo a caso *d*, si costruisce una prima soluzione *S* "a stella" unendo con due archi tutti i nodi a *d* e si calcola $z(S)=(\sum_j c_{dj} +c_{jd})$.

Si calcola la matrice dei savings $s_{ij}=c_{id}+c_{dj}-c_{ij}$ e si ordinano gli s_{ij} in senso decrescente in una lista di indice iniziale $t=1$.

2.Analisi dei savings

A partire dall'elemento *t* si individua il massimo s'_{ij} tale che *i* e *j* siano estremi di circuiti diversi .Si uniscono *i* e *j* e si cancellano gli spigoli *(i,d)* e *(j,d)*; si pone $z(S)=z(S)- s'_{ij}$ e *t'* pari alla posizione di s'_{ij} nella lista dei savings.

3.Criterio di arresto

Se il circuito è unico l'algoritmo si arresta, altrimenti si pone *t=t'* e si ritorna al passo 2.

Terminata la panoramica sugli algoritmi di tipo costruttivo si passa ora alla descrizione della seconda categoria citata ovvero gli algoritmi migliorativi.
Gli algoritmi di ricerca locale si basano sul concetto di intorno di una soluzione, ossia l'insieme di altre soluzioni che si possono ottenere da essa con un'opportuna famiglia di trasformazioni. Ogni trasformazione della famiglia genera una diversa soluzione dell'intorno. Si parla di intorno perché in generale le soluzioni trasformate sono simili a quella di partenza.
Gli algoritmi di ricerca locale partono da una soluzione ottenuta in qualche modo (ad esempio con un algoritmo costruttivo). Quindi generano le soluzioni dell'intorno, cercandone una migliore di quella di partenza. Fra quelle migliori ne scelgono una da sostituire alla soluzione iniziale e ripartono da capo, generando ed esplorando l'intorno di quest'ultima. A seconda degli algoritmi, si sceglie la prima soluzione migliorante che si trova, oppure si generano tutte e poi si sceglie la migliore o si seguono altre strategie ancora.
Il cuore degli algoritmi di ricerca locale è la definizione dell'intorno. Questo deve essere ampio, perché altrimenti non vi si trovano buone soluzioni, ma deve essere anche ristretto perché altrimenti esplorarlo in modo esaustivo diventa troppo pesante.
Il difetto fondamentale di questi algoritmi sta nel fatto che non trovano l'ottimo globale cioè la soluzione migliore fra tutte quelle ammissibili, bensì l'ottimo locale cioè la soluzione migliore fra tutte quelle dell'intorno.
Esistono molti ottimi locali, in genere, e quello che viene generato dall'algoritmo dipende molto dalla soluzione di partenza che gli si fornisce. D'altra parte, non ci sono regole generali per fornire una buona soluzione di partenza. Di solito, migliore è la soluzione di partenza, migliore è l'ottimo locale generato.

Un modo parziale di sfuggire alla dipendenza dalla soluzione iniziale è quello di ripetere l'esecuzione con molte diverse soluzioni di partenzs. Si parla allora di *restart* o *multistart*

Gli algoritmi di ricerca locale classica, una volta giunti in un ottimo locale, si fermano. Benché fuori dell'intorno possano esistere soluzioni migliori, magari di molto, tali algoritmi sono incapaci di trovarle, perché non "vedono" nulla fuori dell'intorno. Questa è la *miopia* degli algoritmi di ricerca locale, che fa da contraltare a quella degli algoritmi costruttivi.

3.2.6.Algoritmo k-opt

L'algoritmo migliorativo più consolidato per il *TSP* è l'algoritmo di scambio ottimale di *k* archi detto anche *k-opt*. A partire da un circuito iniziale, si muovono *k* archi producendo *k* percorsi disgiunti che vengono ricollegati tra loro con l'introduzione di altri *k* archi. Tra tutti i possibili scambi si seleziona quello che consente di ottenere il maggior miglioramento della soluzione.

Chiaramente all'aumento di *k* aumenta necessariamente il tempo di calcolo, però ragionevolmente aumenta anche l'efficacia del singolo passo. Uno dei punti chiave di questo tipo di algoritmi è, infatti, trovare il giusto compromesso tra velocità di calcolo ed efficacia di risoluzione.

Negli anni 50 Lin ha dimostrato che il passaggio da $k = 2$ a $k = 3$ comporta un miglioramento sensibile della qualità della soluzione, mentre il successivo passaggio a $k = 4$ non giustifica in termini di risultato la pesantezza computazionale (infatti, poiché una volta rimossi *k* archi, esistono, nel caso simmetrico $2^{k-1}-1$ modi alternativi di ricomporre il circuito, dal punto di vista computazionale non è possibile utilizzare valori di *k* molto elevati).

Nel caso simmetrico per $k=2$, cancellando una coppia di archi, vi è un solo modo di ricomporre il circuito, mentre per $k=3$ esistono tre possibilità alternative. Nel caso asimmetrico la procedura ha senso solo per $k \geq 3$.

Schematicamente l'algoritmo *k-opt* si articola secondo i seguenti passi:

1.Inizializzazione

Si individua un circuito *S* a caso o con un algoritmo costruttivo.

2.Valutazione degli scambi di k archi

A partire da *S* si individua lo scambio di *k* archi che produce il maggior decremento della lunghezza del circuito.

3.Criterio di arresto

Si ripete il passo 2 fin quando è possibile migliorare la soluzione.

Una variante dell'algoritmo *k-opt* precedentemente descritto è l'algoritmo di Lin-Keringhan. Questo algoritmo è molto simile al precedente; la differenza sta nella variazione di *k* ad ogni singolo passaggio. L'algoritmo esamina per valori crescenti di *k* se uno scambio di *k* archi produce un ciclo più corto del precedente. Ad ogni passaggio il numero *k* viene riazzerato e il procedimento si reitera.

3.3.Metaeuristiche

Anche utilizzando intorni di grande dimensione tutti gli algoritmi di ricerca locale per problemi di ottimizzazione combinatoria si arrestano per aver determinato un ottimo locale rispetto alla funzione intorno utilizzata, o, più in generale, perché non sono in grado di trovare "mosse" che generino soluzioni migliori della soluzione corrente. Qualora ci sia motivo per credere che la soluzione corrente non sia un ottimo globale, si pone quindi il problema di cercare di determinare un diverso, e possibilmente migliore, ottimo locale.
Si discuteranno quindi alcune possibili strategie che permettono di fare questo, cercando di illustrare alcune delle loro principali caratteristiche e potenziali limitazioni. Queste strategie sono chiamate *metaeuristiche* perché non sono algoritmi specifici per un dato problema ma metodi generali che possono essere applicati per tentare di migliorarne le prestazioni di molti diversi algoritmi di ricerca locale.

3.3.1.Multistart

In generale, la qualità dell'ottimo locale determinato da un algoritmo di ricerca locale dipende da due fattori: l'intorno utilizzato e la soluzione ammissibile iniziale da cui la ricerca parte.
Come è noto, spesso è possibile definire più di un algoritmo greedy per lo stesso problema, ed in molti casi gli algoritmi sono anche molto simili, differendo solamente per

l'ordine in cui sono compiute alcune scelte, per cui non è irragionevole pensare di avere a disposizione più di un algoritmo in grado di produrre soluzioni iniziali. In questo caso, le soluzioni prodotte saranno normalmente diverse; inoltre, non è detto che l'ottimo locale determinato eseguendo l'algoritmo di ricerca locale a partire dalla migliore delle soluzioni cosi ottenute sia necessariamente il migliore degli ottimi locali ottenibili eseguendo l'algoritmo di ricerca locale a partire da ciascuna delle soluzioni separatamente.

Tutto ciò suggerisce un'ovvia estensione dell'algoritmo di ricerca locale: generare più soluzioni iniziali, eseguire l'algoritmo di ricerca locale a partire da ciascuna di esse, quindi selezionare la migliore delle soluzioni così ottenute. Questo procedimento è particolarmente attraente quando sia possibile, tipicamente attraverso l'uso di tecniche randomizzate, generare facilmente un insieme arbitrariamente numeroso di soluzioni iniziali potenzialmente diverse.

La combinazione di un algoritmo di ricerca locale e di un'euristica randomizzata viene denominato metodo *multistart*. Normalmente, una volta sviluppata una qualsiasi euristica greedy per determinare la soluzione iniziale ed un qualsiasi algoritmo di ricerca locale per un dato problema, è praticamente immediato combinare le due per costruire un metodo multistart.

Sotto opportune ipotesi tecniche è possibile dimostrare che ripetendo un numero sufficientemente alto di volte la procedura, ossia generando un numero sufficientemente alto di volte soluzioni iniziali, si è praticamente certi di determinare una soluzione ottima del problema. Si noti che, al limite, non è neppure necessaria la fase di ricerca locale: un'euristica randomizzata ripetuta più volte è comunque un modo per generare un insieme di soluzioni ammissibili, e se l'insieme è abbastanza ampio è molto probabile che contenga anche la soluzione ottima. Questo risultato non deve illudere sulla reale possibilità di giungere ad una (quasi) certezza di ottimalità utilizzando un metodo di questo tipo: in generale, il numero di ripetizioni necessarie può essere enorme, in modo tale che risulterebbe comunque più conveniente enumerare tutte le soluzioni del problema.

Le euristiche multistart forniscono quindi un modo semplice, ma non particolarmente efficiente, per cercare di migliorare la qualità delle soluzioni determinate da un algoritmo di ricerca locale.

Anche se non esistono regole che valgano per qualsiasi problema, è possibile enunciare alcune linee guida che si sono dimostrate valide per molti problemi diversi.

In generale, il meccanismo della ripartenza da un punto casuale non fornisce un sostituto efficiente di una ricerca locale ben congegnata: se si confrontano le prestazioni di un metodo multistart che esegue molte ricerche locali con “intorni piccoli" e di uno che

esegue poche ricerche locali con “intorni grandi" è di solito il secondo a fornire soluzioni migliori a parità di tempo totale utilizzato.

Questo comportamento è spiegato dal fatto che una ripartenza da un punto pseudocasuale “cancella la storia" dell'algoritmo: l'evoluzione successiva è completamente indipendente da tutto quello che è accaduto prima della ripartenza. In altri termini, il metodo multistart non è in grado di sfruttare in alcun modo l'informazione generata durante le ricerche locali precedenti per “guidare" la ricerca locale corrente. Ad esempio, è noto che per molti problemi le soluzioni di buona qualità sono normalmente abbastanza “vicine" le une alle altre (in termini di numero di mosse degli intorni utilizzati); quindi, l'aver determinato una “buona" soluzione fornisce una qualche forma di informazione che il metodo multistart non tiene in alcun modo in considerazione. Al contrario, la ricerca locale ha un qualche tipo di informazione sulla “storia" dell'algoritmo, in particolare data dalla soluzione corrente.

La capacità di sfruttare l'informazione contenuta nelle soluzioni precedentemente generate è, in effetti, uno degli elementi che contraddistinguono le metaeuristiche più efficienti, quali quelle discusse nel seguito.

3.3.2.Simulated annealing

L'idea di base del *simulated annealing* è quella di modificare l'algoritmo di ricerca locale sostituendo i criteri deterministici di selezione del nuovo punto nell'intorno corrente e di accettazione della mossa con criteri randomizzati. Uno schema generale di algoritmo di tipo simulated annealing (per un problema di minimo) è il seguente:

```
Procedure Simulated-Annealling(F, c, x):
  begin
    x := Ammissibile(F); x* := x; c := InitTemp();
    while c ≥ c̄ do
      for i := 1 to k(c) do
        ⟨ seleziona x' ∈ I(x) in modo pseudocasuale ⟩;
        if( c(x') < c(x*) ) then x* := x';
        if( c(x') < c(x) )
          then x := x';                    /* downhill */
          else begin
            r := random(0, 1);
            if( r < e^{-(c(x')-c(x))/c} )
              then x := x';                /* uphill */
          end
      endfor
      ⟨ decrementa c ⟩;
    endwhile
  x := x*;
  end.
```

L'algoritmo prende spunto da un metodo usato in pratica per produrre cristalli con elevato grado di regolarità e per questo è usuale chiamare il parametro c “temperatura”. L'idea alla base dell'algoritmo è quella di permettere consistenti peggioramenti del valore della funzione obiettivo nelle fasi iniziali dell'esecuzione, in modo da evitare di rimanere intrappolati in ottimi locali molto “lontani" dall'ottimo globale. Dopo un numero sufficiente di iterazioni l'algoritmo dovrebbe aver raggiunto una parte dello spazio delle soluzioni “vicine" all'ottimo globale: a quel punto la “temperatura” viene diminuita per raffinare la ricerca.

In particolare, si può dimostrare che esiste una costante *C* tale che se la temperatura decresce non più rapidamente di *C=log(k)*, dove *k* è il numero di iterazioni compiute, allora l'algoritmo determina una soluzione ottima con probabilità pari a uno. Come nel caso del multistart, però, queste proprietà teoriche non hanno grande utilità in pratica: la regola di raffreddamento sopra indicata corrisponde ad una diminuzione molto lenta della temperatura, e quindi ad un numero di iterazioni talmente grande da rendere più conveniente l'enumerazione di tutte le soluzioni ammissibili. In pratica si usano quindi regole di raffreddamento esponenziali in cui la temperatura viene moltiplicata per un fattore *c<1* dopo un numero fissato di iterazioni, e la temperatura iniziale *c* viene scelta come la maggior differenza di costo possibile tra due soluzioni *x* ed x_0 tali che $x_0 \in I(x)$ (in modo tale da rendere possibile, almeno inizialmente, qualsiasi mossa). In questo modo non si ha alcuna certezza, neanche in senso stocastico, di determinare una soluzione ottima, ma si possono ottenere euristiche di buona efficienza in pratica.

Gli algoritmi di tipo simulated annealing hanno avuto un buon successo in alcuni campi di applicazioni dell'Ottimizzazione Combinatoria e la ragione di questo successo risiede nel fatto che sono relativamente semplici da implementare, poco dipendenti dalla struttura del problema e quindi facilmente adattabili a problemi diversi ed abbastanza “robusti", nel senso che, se lasciati in esecuzione sufficientemente a lungo, solitamente producono soluzioni di buona qualità in quasi tutti i problemi in cui sono applicati. Un vantaggio in questo senso è il fatto che contengano relativamente pochi parametri, ossia la temperatura iniziale, il valore di *c* e la durata di ciascuna fase. Ogniqualvolta il comportamento di un algoritmo dipende da parametri che possono essere scelti in modo arbitrario (sia pur seguendo certe regole) si pone, infatti, il problema di determinare un valore dei parametri che produce soluzioni di buona qualità quanto più rapidamente possibile per le istanze che si intende risolvere. Per fare questo è normalmente necessario eseguire l'algoritmo molte volte su un nutrito insieme di istanze di test con combinazioni diverse dei parametri, in modo da ottenere informazione sull'impatto dei diversi parametri sull'efficacia ed efficienza dell'algoritmo. Questo processo può richiedere molto tempo, specialmente se l'algoritmo è poco “robusto", ossia il suo comportamento varia considerevolmente per piccole variazioni dei parametri o, con gli stessi parametri, per istanze simili. Il simulated annealing risulta piuttosto “robusto" da questo punto di vista, anche per la disponibilità di linee guida generali per l'impostazione dei parametri che si sono rivelate valide in molti diversi contesti.

In linea generale, si può affermare che normalmente la tecnica del simulated annealing permette di migliorare la qualità delle soluzioni fornite da un algoritmo di ricerca locale con una ragionevole efficienza complessiva. Il metodo risulta molto spesso più efficien-

te dei metodi multistart basati sullo stesso intorno, poiché mantiene una maggiore quantità di informazione sulla storia dell'algoritmo (non solo la soluzione corrente, ma anche la temperatura) che in qualche modo “guida" la ricerca di una soluzione ottima.
Questo tipo di tecnica risulta particolarmente adatta a problemi molto complessi, di cui sia difficile sfruttare la struttura combinatoria, e nel caso in cui l'efficacia del metodo (la qualità della soluzione determinata) sia più importante della sua efficienza (la rapidità con cui la soluzione è determinata).
In casi diversi altri tipi di tecniche, come quelle discusse nel seguito, possono risultare preferibili; questo è giustificato dal fatto che l'informazione sulla storia dell'algoritmo mantenuta da un algoritmo di tipo simulated annealing è molto “aggregata", e quindi non particolarmente efficiente nel guidare la ricerca. Inoltre, il simulated annealing si basa fondamentalmente su decisioni di tipo pseudo-casuale, il che tipicamente produce un comportamento “medio" affidabile ma difficilmente più efficiente di decisioni deterministiche guidate da criteri opportuni. Infine, la generalità del metodo, ossia il fatto che l'algoritmo sia largamente indipendente dal problema, implica che il metodo non sfrutta appieno la struttura del problema; la capacità di sfruttare quanto più possibile tale struttura è uno dei fattori fondamentali che caratterizzano gli algoritmi più efficienti.

3.3.3.Ricerca taboo

Un problema fondamentale che deve essere superato quando si intende modificare un algoritmo di ricerca locale consiste nel fatto che accettare mosse che peggiorano il valore della funzione obiettivo (“uphill") pone ad immediato rischio di ritornare sulla precedente soluzione corrente, e quindi di entrare in un ciclo in cui si ripetono sempre le stesse soluzioni.
E’ quindi necessario porre in atto tecniche che impediscano, o comunque rendano altamente improbabile, l'entrata in un ciclo. La più diffusa da queste tecniche è quella delle mosse Taboo, che caratterizza un'ampia famiglia di algoritmi detti di *ricerca Taboo*. Questi algoritmi sono, in prima approssimazione, normali algoritmi di ricerca locale, in cui cioè si compiono mosse di “downhill" fin quando ciò sia possibile; quando però il punto corrente è un ottimo locale per l'intorno utilizzato, e quindi un normale algoritmo di ricerca locale terminerebbe, un algoritmo di ricerca Taboo seleziona comunque una soluzione $x' \in I(x)$ secondo criteri opportuni e compie una mossa “uphill". Per evitare i cicli, l'algoritmo mantiene una lista Taboo che contiene una descrizione delle mosse “uphill"; ad ogni iterazione, nel cercare una soluzione $x' \in I(x)$ con $c(x') < c(x)$, l'algoritmo controlla la lista Taboo, e scarta tutte le soluzioni che sono generate da una mossa Taboo. Siccome l'algoritmo permette mosse “uphill", non è garantito che la soluzio-

ne corrente al momento in cui l'algoritmo termina sia la migliore determinata; come nel caso del simulated annealing, si mantiene quindi, oltre alla soluzione corrente, la miglior soluzione x^* tra tutte quelle determinate.

In prima approssimazione si potrebbe pensare che la lista Taboo contenga tutte le soluzioni precedentemente trovate con valore della funzione obiettivo migliore di quello della soluzione corrente; questo chiaramente eviterebbe i cicli, ma normalmente risulta eccessivamente costoso, sia in termini di memoria richiesta che in termini del costo di verificare che una data soluzione appartenga alla lista. In generale si preferisce quindi mantenere nella lista Taboo una descrizione, anche parziale, delle mosse (che definiscono la funzione intorno) che hanno generato la soluzione corrispondente.

Questa descrizione dipende quindi dalla particolare funzione intorno utilizzata, e deve essere “compatibile" con la funzione σ, ossia non deve rendere “eccessivamente più costosa" la determinazione della nuova soluzione. In effetti, per semplificare il controllo della lista Taboo non è infrequente che si implementi una lista Taboo che contiene una descrizione parziale delle mosse utilizzate.

Inserire mosse nella lista Taboo diminuisce il numero di soluzioni considerate ad ogni passo come possibili nuove soluzioni correnti: questo può seriamente limitare la capacità dell'algoritmo di scoprire soluzioni migliori.

Per questo sono necessari dei meccanismi che limitino l'effetto della lista Taboo.

Il primo meccanismo, usato in tutti gli algoritmi di ricerca Taboo, consiste semplicemente nel limitare la massima lunghezza della lista Taboo ad una costante fissata: quando la lista ha raggiunto tale lunghezza ed una nuova mossa deve essere resa Taboo, la “più vecchia" delle mosse nella lista viene nuovamente resa ammissibile. In pratica gli algoritmi di tipo ricerca Taboo non entrano in ciclo se la lista Taboo contiene anche solo poche decine di mosse. Ciò dipende dal fatto che il numero di possibili soluzioni che possono essere visitate con k mosse di ricerca locale aumenta esponenzialmente in k, e quindi è molto poco probabile che l'algoritmo visiti esattamente, tra tutte quelle possibili, una delle soluzioni già generate.

Un secondo metodo per limitare gli effetti negativi della lista Taboo è quello di inserire un cosiddetto criterio di aspirazione, ossia un insieme di condizioni logiche tale per cui se la soluzione $x' \in I(x)$ generata da una mossa soddisfa almeno una di queste condizioni, allora x' può essere considerata tra i possibili candidati a divenire la prossima soluzione corrente, anche se la mossa appartiene alla lista Taboo. Ovviamente, il criterio di aspirazione deve essere scelto in modo tale da garantire (o rendere molto probabile) che l'algoritmo non entri in un ciclo.

Oltre a queste componenti base, gli algoritmi di ricerca Taboo spesso contengono altre strategie che risultano utili per rendere il processo di ricerca più efficiente. Ad esempio,

per quei problemi in cui le “buone" soluzioni sono spesso “vicine" spesso si utilizzano tecniche di intensificazione, che concentrano temporaneamente la ricerca “vicino" alla soluzione corrente.

Questo può essere ottenuto ad esempio restringendo la dimensione dell'intorno, oppure modificando la funzione obiettivo con un termine che penalizza leggermente le soluzioni dell'intorno più “lontane" da quella corrente. L’intensificazione viene solitamente effettuata quando si determina una nuova miglior soluzione x^*, e mantenuta per un numero limitato di iterazioni, nella speranza che vicino alla migliore soluzione determinata fino a quel momento ci siano altre soluzioni ancora migliori.

Una strategia in un certo senso opposta è quella della diversificazione, che cerca di indirizzare la ricerca verso aree dello spazio delle soluzioni diverse da quelle esplorate fino a quel momento. Modi possibili per implementare strategie di diversificazione sono ad esempio penalizzare leggermente le soluzioni dell'intorno “vicine" alla soluzione corrente x, allargare la dimensione dell'intorno, effettuare in una sola iterazione combinazioni di molte mosse (ad esempio selezionandole in modo randomizzate) per “saltare" in una zona dello spazio delle soluzioni “lontana" da x, o persino effettuare una ripartenza pseudo-casuale come nel metodo multistart. Normalmente, vengono effettuate mosse di diversificazione quando per “molte" iterazioni successive la migliore soluzione trovata x^* non cambia, suggerendo che l'area dello spazio delle soluzioni in cui si trova la soluzione corrente sia già stata sostanzialmente esplorata, e che sia quindi necessario visitarne una diversa.

Nell'implementare tecniche di intensificazione e diversificazione risulta spesso utile sfruttare l'informazione contenuta nella sequenza di soluzioni generate, ad esempio associando a ciascun costituente elementare delle soluzioni un qualche valore numerico che ne rappresenti in qualche modo “l'importanza".

Per risolvere il *TSP* si pensi ad esempio di applicare un qualche algoritmo di ricerca locale, e di mantenere per ogni arco un contatore che indichi di quante soluzioni correnti l'arco ha fatto parte fino a quel momento: archi con un valore alto del contatore sono in qualche senso “più rilevanti" di archi con un valore basso del contatore. In una procedura di diversificazione potrebbe essere ragionevole concentrarsi su archi con valore alto del contatore, in modo da esplorare soluzioni “diverse" da quelle visitate fino a quel momento.

Infine, una strategia che talvolta si rivela utile è quella di permettere che la soluzione corrente x diventi (per un numero limitato di iterazioni) inammissibile, se questo serve a migliorare sensibilmente il valore della soluzione obiettivo. Questo viene fatto per i problemi in cui sia particolarmente difficile costruire soluzioni ammissibili: le poche" soluzioni ammissibili” del problema possono quindi essere “lontane", ed è necessario per-

mettere mosse in cui si perde l'ammissibilità, seguite da una successione di mosse in cui si cerca di riguadagnarla.
La grande flessibilità data dai molti parametri, ed il fatto che ogni aspetto di un algoritmo di ricerca Taboo debba essere adattato allo specifico problema di ottimizzazione combinatoria risolto, sono però anche le caratteristiche che possono rendere questi algoritmi molto efficienti in pratica.

3.3.4.Algoritmi genetici

Gli algoritmi genetici, che cercano di riprodurre alcuni dei meccanismi che si ritiene stiano alla base dell'evoluzione delle forme di vita, hanno la caratteristica peculiare di mantenere non una singola soluzione corrente ma una popolazione di soluzioni. L'algoritmo procede per fasi, corrispondenti a “generazioni" nella popolazione. In ogni fase vengono ripetute un numero opportuno di volte le seguenti operazioni:

• all'interno della popolazione vengono selezionate in modo pseudo-casuale due o più soluzioni “genitrici";

• a partire da tali soluzioni vengono generate in modo pseudo-casuale un certo numero di soluzioni “discendenti", ottenute “mescolando" le caratteristiche di tutte le soluzioni genitrici;

• a ciascuna soluzione così generata vengono applicate alcune “mutazioni casuali" che cercano di introdurre nella popolazione caratteristiche altrimenti non presenti.

Alla fine di ogni fase si procede alla selezione: a ciascuna delle soluzioni, sia quelle presenti nella popolazione all'inizio della fase che quelle generate durante la fase, viene associato un valore di adattamento (fitness), ad esempio il valore della funzione obiettivo, e vengono mantenute nella popolazione (sopravvivono alla generazione successiva) solo le soluzioni con miglior fitness, mantenendo costante la dimensione della popolazione; alternativamente, le soluzioni sopravvissute sono selezionate in modo pseudo-casuale con probabilità dipendente dal fitness.
Tutte le operazioni di un algoritmo genetico devono essere specializzate per il problema da risolvere, anche se per alcune esistono implementazioni abbastanza standard. Ad esempio, la selezione delle soluzioni genitrici viene normalmente effettuata semplicemente estraendo in modo casuale dalla popolazione; la probabilità di essere estratta

può essere uniforme oppure dipendere dal valore di fitness. Analogamente, la selezione alla fine di ogni fase è normalmente effettuata, in modo deterministico o pseudo-casuale, semplicemente privilegiando le soluzioni con miglior fitness. Per quanto riguarda la definizione del valore di fitness, spesso si usa semplicemente il valore della funzione obiettivo (in alcuni casi può risultare opportuno modificare la funzione obiettivo per premiare soluzioni che, a parità di funzione obiettivo, appaiano più desiderabili).
La generazione di soluzioni e le mutazioni casuali, invece, sono strettamente dipendenti dal problema. Questo non è stato inizialmente ben compreso: dato che qualsiasi soluzione ammissibile rappresentata in un computer può essere vista come una stringa di bits, si è sostenuto che le operazioni di generazione e mutazione potessero essere implementate in modo generico, ad esempio rimpiazzando nel “patrimonio genetico" di un genitore una o più sottostringhe di bits con quelle prelevate dalle corrispondenti locazioni nel “patrimonio genetico" dell'altro genitore, e cambiando il valore di un piccolo numero dei bits scelti in modo pseudo-casuale. Questo risulta, in effetti, possibile per problemi con “pochi vincoli", in cui sia molto facile costruire una soluzione ammissibile.
Nella maggioranza dei casi questo però non risulta effettivamente possibile ed è quindi necessario sviluppare euristiche specifiche che costruiscano soluzioni ammissibili cercando di replicare, per quanto possibile, le caratteristiche di entrambe i genitori.
Nel caso del *TSP* ad esempio dati due cicli hamiltoniani diversi, si devono produrre uno o più cicli che “combinino" le caratteristiche dei due genitori. Un possibile modo di procedere è quello di considerare fissati, nei cicli “figli", tutti gli archi che sono comuni ad entrambe i genitori, e poi cercare di completare questa soluzione parziale fino a formare un ciclo hamiltoniano mediante un'euristica costruttiva randomizzata analoga alla nearest neighbour. Definito un nodo “corrente" iniziale *i*, l'euristica prosegue sicuramente sul nodo *j* se l'arco *(i,j)* appartiene ad entrambe i genitori. Se invece il “successore" di *i* nei due cicli hamiltoniani è diverso, l'euristica seleziona uno dei due successori che non sia ancora stato visitato secondo un qualche criterio opportuno; ad esempio, quello a cui corrisponde l'arco di costo minore, oppure in modo pseudo-casuale con probabilità uniforme, oppure con probabilità dipendente dal costo dell'arco, oppure ancora con probabilità dipendente dal valore della funzione obiettivo del “genitore" a cui l'arco corrispondente appartiene (favorendo le scelte del “genitore migliore"). Qualora entrambi i successori siano stati già visitati l'euristica selezionerà un diverso nodo secondo altri criteri (ad esempio quello “più vicino”); se tutti i nodi raggiungibili da *i* sono già visitati (e non si è formato un ciclo) l'euristica fallirà, “abortendo" la generazione della nuova soluzione.
Le mutazioni casuali sono normalmente più facili da realizzare, soprattutto qualora si disponga già di un algoritmo di ricerca locale per il problema. Infatti, un modo tipico per

realizzare mutazioni è quello di effettuare un piccolo numero di mosse scelte in modo pseudo-casuale, a partire dalla soluzione generata. Nel caso del *TSP* si potrebbero effettuare un piccolo numero di 2-scambi tra coppie *(i,j)* e *(h,k)* di archi del ciclo selezionate in modo pseudo-casuale.

Infine, qualsiasi algoritmo genetico richiede una prima fase in cui viene generata la popolazione iniziale. Ciò può essere ottenuto in vari modi: ad esempio, si può utilizzare un'euristica randomizzata analogamente a quanto visto per il metodo multistart. Alternativamente si può utilizzare un algoritmo di ricerca locale, eventualmente con multistart, tenendo traccia di un opportuno sottoinsieme delle soluzioni visitate, che costituiranno a terminazione la popolazione iniziale per l'algoritmo genetico.

Anche gli algoritmi genetici, come quelli di ricerca Taboo, dipendono da un numero di parametri il cui valore può non essere facile da fissare. Un parametro molto importante è ad esempio la cardinalità della popolazione: popolazioni troppo piccole non permettono di diversificare a sufficienza la ricerca, che può rimanere “intrappolata" in una zona dello spazio delle soluzioni, mentre una popolazione troppo numerosa può rendere poco efficiente l'algoritmo. Altri parametri importanti riguardano il numero di nuove soluzioni costruite ad ogni generazione, i parametri per la scelta dei genitori e la selezione e così via.

Gli algoritmi genetici “puri", ossia che seguono lo schema precedente, possono rivelarsi efficienti ed efficaci; spesso, però, algoritmi basati sulla ricerca locale risultano maggiormente efficienti. In effetti, gli algoritmi genetici di maggior successo sono normalmente quelli che integrano entrambe le tecniche: ad ogni soluzione generata, dopo la fase di mutazione, viene applicata una procedura di ricerca locale per tentare di migliorarne il valore di fitness. Un algoritmo di questo tipo può anche essere considerato un metodo “multistart con memoria", in cui cioè le soluzioni di partenza del metodo multistart non sono selezionate in modo completamente pseudo-casuale, ma si cerca di utilizzare l'informazione precedentemente generata.

La combinazione degli algoritmi genetici e degli algoritmi di ricerca locale può assumere molte forme diverse. I casi estremi corrispondono all'algoritmo genetico “puro", in cui cioè non si effettua nessuna ricerca locale, ed all'algoritmo di ricerca locale “puro", in cui cioè la popolazione è formata da un solo individuo. I metodi ibridi si differenziano fondamentalmente per la quantità di risorse che vengono spese nell'uno e nell'altro tipo di operazione: un algoritmo in cui la ricerca locale sia molto semplice, ed eventualmente eseguita al più un numero fissato di mosse, avrà un comportamento più simile a quello di un algoritmo genetico “puro", mentre un algoritmo in cui la ricerca locale sia effettuata in modo estensivo (ad esempio con tecniche Taboo e con intorni di grande

dimensione) avrà un comportamento più simile a quello di un algoritmo di ricerca locale “puro" con multistart.

Esiste inoltre la possibilità di eseguire sequenzialmente un algoritmo genetico ed uno di ricerca locale, eventualmente più volte: l'algoritmo di ricerca locale fornisce la popolazione iniziale per quello genetico e questi fornisce una nuova soluzione iniziale per la ricerca locale una volta che questa si sia fermata in un ottimo locale.

Dato che esistono moltissimi modi per combinare i due approcci, è molto difficile fornire linee guida generali; ciò è particolarmente vero in quando in un approccio ibrido è necessario determinare il valore di molti parametri (per entrambe gli algoritmi e per la loro combinazione), rendendo la fase di messa a punto dell'algoritmo potenzialmente lunga e complessa. Solo l'esperienza può quindi fornire indicazioni sulla migliore strategia da utilizzare per un determinato problema di ottimizzazione. Si può però affermare che un'opportuna combinazione delle tecniche descritte in questo capitolo può permettere di determinare efficientemente soluzioni ammissibili di buona qualità per moltissimi problemi di ottimizzazione, per quanto al costo di un consistente investimento nello sviluppo e nella messa a punto di approcci potenzialmente assai complessi.

Conclusioni

Con tale lavoro si è voluto offrire al lettore una panoramica generale, di natura prettamente compilativa, riguardante il problema del commesso viaggiatore nella sua formulazione più generale, ovvero il caso asimmetrico, e delle principali tecniche euristiche per la risoluzione del suddetto. Come evidenziato nella trattazione il TSP rientra nella classe dei problemi di ottimizzazione combinatoria, ossia di quei problemi affrontati nell'ambito della Ricerca Operativa che, pur presupponendo ad un numero finito di soluzioni all'interno delle quali vi è con certezza l'ottimo ricercato, risultano spesso avere una complessità computazionale tale da non permettere il raggiungimento di un risultato soddisfacente in un tempo di attesa "ragionevole". Proprio in questa ottica le tecniche euristiche descritte nella trattazione, pur mantenendo tutti i limiti legati alla loro stessa natura di tecniche "approssimative", permettono, a fronte dell'ottenimento di una soluzione che non rappresenta l'ottimo esatto del problema preso in esame, di avere una "buona soluzione" (dove con tale espressione si intende un risultato con un valore accettabile, nel range che ci si prefigge, del valore della funzione obbiettivo) del problema considerato e in un tempo ragionevole per gli scopi che ci si prefigge. Proprio per questa loro caratteristica di praticità le tecniche euristiche di ottimizzazione assumono una grande rilevanza dal punto di vista operativo, venendo sempre più impiegate in un'ampia gamma di situazioni che si verificano nella pratica lavorativa e che possono essere modellizzate con le formulazioni matematiche in precedenza descritte. Una vasta gamma di problemi di logistica distributiva, come la movimentazione di pezzi tra reparti produttivi, la raccolta e distribuzione dei materiali, la distribuzione di merci da centri di produzione a centri di distribuzione, trovano sempre più spesso la loro risoluzione nell'applicazione ai casi reali di tecniche euristiche di ottimizzazione che danno la possibilità, grazie alla loro natura, di arrivare ad una buona soluzione del problema in tempi relativamente brevi e con costi contenuti.

Bibliografia

Bruno G. (2003). *Operations Management: modelli e metodi per la logistica.* Napoli: Edizioni Scientifiche Italiane.

Ciriani A. (2001, vol.VI, n.4). "4>15112". *AIRO news* , p. 9-10.

Colorini A. (1988). *Elementi di ricerca operativa: introduzione ai modelli matematici di decisione.* Bologna: Zanichelli.

Fischetti M. (1995). *Lezioni di Ricerca Operativa.* Padova: Libreria Progetto Padova.

Frego M. , Pizzato M. (s.d.). *Il problema del commesso viaggiatore - The travelling salesman problem.* Tratto il giorno Maggio 14, 2011 da Webalice: http://www.webalice.it/m.frego/MagnaCharta/Tsp.pdf

Gallo G. *Algoritmi euristici.* Tratto il giorno Maggio 25, 2011 da Dipartimento di informatica - Università di Pisa: www.di.unipi.it/optimize/Courses/OCR/0910/Appunti/Capitolo5

Monaci M. *Problema del commesso viaggiatore.* Tratto il giorno Maggio 15, 2011 da Dipartimento di ingegneria dell'Informazione - Università degli studi di Padova: http://www.dei.unipd.it/~monaci/tsp_rev30.pdf

RomaninJacur G., Andreatta G., Mason M. (1990). *Appunti di ottimizzazione su reti.* Padova: Edizioni Libreria Progetto Padova.

Romano P., Danese P. (2006). *Supply chain management: la gestione dei processi di fornitura e distribuzione.* Milano: McGraw-Hill.

Vercellis C. (1997). *Modelli e decisioni: strumenti e metodi per le decisioni aziendali.* Bologna: Progetto Leonardo.