

Algoritmi Euristici

Manuel Pagliuca

17 dicembre 2021

INDICE

Indice

1	Introduzione	7
2	Algoritmi euristici	7
2.1	Cenni sulla parola	7
2.2	Definizione	7
2.3	Motivazioni per gli algoritmi euristici	8
3	Classificazione dei problemi	8
3.1	Problemi di ottimizzazione e ricerca	8
3.2	perché siamo interessati nei problemi di ottimizzazione e ricerca ?	9
3.3	Problema di ottimizzazione combinatoria	10
3.4	Euristiche basate sui modelli	10
3.5	Definizione alternativa di CO	11
3.6	Euristiche basate su soluzioni	11
3.6.1	Randomizzazione e memoria	12
3.7	Rischi da cui stare attenti	12
4	Problemi di ottimizzazione combinatoria	13
4.1	Insieme dei problemi pesati	13
4.1.1	Knapsack Problem (KP)	13
4.1.2	Maximum Diversity Problem (MDP)	15
4.1.3	Interludio 1: la funzione obiettivo	17
4.2	Insieme dei problemi di partizionamento	18
4.2.1	Bin Packing Problem (BPP)	18
4.2.2	Parallel Machine Scheduling Problem (PMSP)	20
4.2.3	Interludio 2: la funzione obiettivo, ancora	22
4.3	Problemi delle funzioni logiche	23
4.3.1	The Max-SAT problem	23
4.4	Problemi con matrici numeriche	25
4.4.1	Set covering problem (SCP)	25
4.4.2	Interludio 3: il test di fattibilità	26
4.5	Set Packing Problem	27
4.6	Set Partitioning Problem	28
4.7	Interludio 4: ricerca di soluzioni fattibili	30
4.8	Problemi sui grafi	31
4.8.1	Vertex Cover Problem	31
4.8.2	Maximum Clique Problem (MCP)	32
4.8.3	Maximum Independent Set	33
4.8.4	Interludio 5: le relazioni tra i problemi	35
4.8.5	Travelling Salesman Problem (TSP)	37
4.8.6	Minimum Capacitated Spanning Tree Problem (MCSTP)	39
4.8.7	Vehicle Routing Problem (VRP)	41
4.8.8	Interludio 6: combinare rappresentazioni alternative	41

INDICE

5 Efficienza teorica	42
5.1 Problemi	42
5.2 Algoritmi	43
5.3 Costi di un algoritmo euristico	43
5.3.1 Il tempo	44
5.3.2 Complessità asintotica nel caso peggiore	44
5.3.3 Gli spazi funzionali Θ	44
5.3.4 Gli spazi funzionali O	45
5.3.5 Gli spazi funzionali Ω	46
5.3.6 L'algoritmo esaustivo	47
5.3.7 Complessità polinomiale ed esponenziale	47
5.3.8 Problemi di trasformazione e riduzione	47
5.4 Complessità parametrizzata	48
5.4.1 Bounded tree search (algoritmo con parametro)	49
5.4.2 Kernelization - "Problem reduction" (algoritmo con pa- rametro)	52
5.5 Complessità nel caso medio (average-case complexity)	55
5.6 Modelli probabilistici per matrici numeriche	55
5.7 Modelli probabilistici per grafi	56
5.8 Modelli probabilistici per funzioni logiche	56
5.9 Transizione di fase	57
5.10 Transizione di fare per 3-SAT e Max-3-SAT	57
5.11 La transizione di fase per il VCP	58
5.12 Costo computazionale degli algoritmi euristici	59
6 Efficacia teorica	59
6.1 Efficacia di un algoritmo euristico	59
6.2 Distanza tra le soluzioni	60
6.2.1 Differenza assoluta	60
6.2.2 Differenza relativa	60
6.2.3 Rapporto di approssimazione	61
6.3 Analisi teorica: garanzia di approssimazione	61
6.3.1 Come ottenere una garanzia di approssimazione?	62
6.4 Inapprossimabilità	68
6.5 Schemi di approssimazione	68
6.6 Oltre al caso peggiore	69
6.6.1 Parametrizzazione	69
6.6.2 Caso medio	69
6.6.3 Randomizzazione	69
7 Valutazione empirica delle prestazioni	70
7.1 Analisi sperimentale	71
7.1.1 Modello	71
7.1.2 Benchmark	71
7.2 Confronto di algoritmi euristici	72
7.2.1 Descrivere le prestazioni di un algoritmo	73

INDICE

7.3	Valutazione a posteriori dell'efficienza	73
7.3.1	Analisi del tempo computazionale (<i>RTD diagram</i>)	73
7.3.2	Analisi del tempo computazionale (<i>scaling diagram</i>)	74
7.3.3	Interpolazione dello <i>scaling diagram</i>	75
7.4	Valutazione a posteriori dell'efficacia	75
7.4.1	Analisi della qualità della soluzione (<i>SQD diagram</i>)	75
7.4.2	Diagrammi SQD parametrici	77
7.4.3	Confronto tra algoritmi utilizzando SQDs	77
7.5	Descrizione statistica compatta	78
7.5.1	Boxplots	79
7.5.2	Relazione tra qualità e tempo computazionale	81
7.5.3	Classificazione	81
7.6	Diagrammi complessi	83
7.6.1	La probabilità di successo	83
7.6.2	Qualified Run time Distribution diagram (QRTD)	83
7.6.3	Timed Solution Quality Distribution diagram (TSQD)	84
7.6.4	Solution Quality Statistics over Time diagram (SQST)	85
7.7	Wilcoxon's test	85
7.7.1	Ipotesi sul Wilcoxon's test	86
7.7.2	Calcolo di p	87
8	Euristiche costruttive	88
8.1	Introduzione alle euristiche costruttive	89
8.1.1	Il grafo di costruzione	89
8.2	Condizione di terminazione	93
8.3	Struttura di algoritmi euristiche costruttivi	94
8.4	Definizione del grafo di costruzione	94
8.5	Il problema dello zaino frazionario (FKP)	95
8.6	Il problema dello zaino (KP)	97
8.7	Maximum Diversity Problem	97
8.7.1	Travelling Salesman Problem	98
8.8	Riassumendo	100
8.9	Una caratterizzazione nel caso additivo	100
8.10	Greedoidi	101
8.11	Matroidi	101
8.11.1	Matroide uniforme per KP/FKP	102
8.11.2	Matroide grafico MST (Minimum Spanning Tree)	103
8.11.3	Matroide per TSP	103
8.11.4	Greedoidi con forti assiomi di scambio : MST	104
8.12	Algoritmi non esatti	104
8.13	Algoritmi costruttivi puri ed adattivi	108
8.14	Euristica First-Fit	111
8.15	Estensione delle euristiche costruttive	112
8.15.1	Euristica di distanza per Stainer Tree Problem (STP)	113
8.15.2	Algoritmi di inserzione per il TSP	116
8.15.3	Efficacia ed efficienza	126

INDICE

8.15.4	Caratteristiche generali degli algoritmi costruttivi	127
8.15.5	Quando sono usati gli algoritmi costruttivi	127
8.15.6	Euristiche distruttive	127
8.15.7	Estensione degli algoritmi costruttivi	129
8.15.8	Euristiche costruttive regret-based	129
8.15.9	Euristica costruttiva roll-out	131
8.16	Metaeuristiche costruttive	132
8.16.1	Condizione di terminazione	133
8.16.2	Multi-start	133
8.16.3	Principali metaeuristiche costruttive	134
8.16.4	Adaptive Research Techinique	135
8.16.5	Semi-greedy heuristics	137
8.16.6	Semi-greedy e GRASP	139
8.16.7	Algoritmo semi-greedy reattivo	142
8.16.8	Metodi di perturbazione dei costi	142
8.16.9	Ottimizzazione della colonia di formiche	143
8.16.10	Varianti dell'Ant System	146
9	Euristiche di scambio	148
9.1	Algoritmi di scambio	148
9.1.1	Vicinato basato sulla distanza	149
9.1.2	Vicinato basato sulle operazioni	149
9.1.3	Vicinato basato sulla distanza e sulle operazioni	150
9.1.4	Connettività del grafo di ricerca	152
9.1.5	Euristica Hill-climbing	153
9.1.6	Proprietà del grafo di ricerca	154
9.1.7	Il paesaggio	156
9.2	Complessità Steepest Descent	160
9.2.1	Esplorazione del vicinato	160
9.2.2	Calcolare o aggiornare la funzione obiettivo: caso additivo	161
9.2.3	Calcolare o aggiornare la funzione obiettivo: caso quadratico	162
9.2.4	Aggiornamento della funzione obiettivo: esempi non linear	163
9.2.5	Schema generale di esplorazione sofisticata	166
9.2.6	Salvataggio parziale del vicinato	167
9.2.7	Trade-off tra efficienza ed efficacia	168
9.2.8	Ritocchi dei vicinati	169
10	Ricerca di vicinato su larga scala	169
10.1	Visita efficiente di vicinati esponenziali	170
10.1.1	Dynasearch	170
10.1.2	Scambi ciclici	171
10.1.3	Grafo di miglioramento	172
10.1.4	Ricerca per il circuito di costo minimo	173
10.1.5	Catene di scambio non cicliche	173

INDICE

10.1.6 Order-first split-second	175
10.1.7 Grafo ausiliario	175
10.1.8 Ricerca in profondità variabile	176
10.1.9 Algoritmo Lin-Kernighan (TSP simmetrico)	179
10.1.10 Dettagli implementativi	182
10.1.11 Metodi iterati golosi (destroy-and-repair)	183
10.2 Multi-start, ILS, e VNS	183
10.2.1 Superare gli ottimi locali	183
10.2.2 Condizione di termine	184
10.2.3 Modificare la soluzione di partenza: generazione casuale	184
10.2.4 Modificando la soluzione di partenza: procedure costruttive	185
10.2.5 Influenza di una soluzione di partenza	186
10.2.6 Iterated Local Search (ILS)	187
10.2.7 Procedura di perturbazione	188
10.2.8 Condizione di accettazione	189
10.3 Variable Neighbourhood Search (VNS)	190
10.3.1 Meccanismo di perturbazione adattivo	191
10.3.2 Schema generale	191
10.3.3 Regolazione dei parametri	192
10.3.4 VNS Distorto (skewed)	192
10.4 VND e DLS	193
10.4.1 Estendere la ricerca locale senza peggiorarla	193
10.4.2 VND	193
10.4.3 DLS	195
11 Metauristiche di ricombinazione	199
11.1 Euristiche di ricombinazione	199
11.2 Schema generale	200
11.3 Scatter Search	200
11.4 Procedura di ricombinazione	202
11.5 Path Relinking	203
11.5.1 Relinking paths	204

1 Introduzione

L'obiettivo di questo corso è quello di mostrare che gli **algoritmi euristici** non sono ricette per problemi specifici: le euristiche e i problemi possono essere combinati liberamente. Una qualsiasi euristica può essere utilizzata su un qualsiasi problema. *Euristica* è una parola che deriva dal Greco, e sta per *"metodologia di ricerca della verità o dei fatti"*, deriva da *heurisko* che significa *io trovo*.

Il termine deriva da una storia molto famosa di Archimede, che aveva da risolvere il problema di imparare se la data corona d'oro era effettivamente d'oro o se fosse solo placcata in oro. Questo era possibile da dedurre conoscendo il rapporto tra peso e volume, però il peso era facile da ottenere, mentre il volume è molto più difficile.

L'idea di Archimede consisteva nell'utilizzo di un secchio colmo di acqua ed immersando la corona in quest'ultimo, il quantitativo di acqua che esce dal secchio consiste nel volume della corona.

2 Algoritmi euristici

2.1 Cenni sulla parola

La parola *euristica* può cambiare di significato in base al campo in cui viene utilizzata, in alcuni settori scientifici la parola *algoritmo euristico* è considerata come **osimoro** in quanto le due parole vengono considerate opposti.

- **Algoritmo** ha un significato che sta per *procedura deterministica e formale*, la quale consiste in una sequenza finita di step elementari.
- **Euristico** ha un significato che sta per *informale, creativo e metodo a "regola aperta"* per trovare una soluzione.

Ogni algoritmo ha (od è) una **dimostrazione di correttezza** mentre un algoritmo euristico non ne ha nessuna. Essi sono trasformazioni meccanico-simboliche che partono da uno *starting point* (chiamato *ipotesi*) e giungono ad un *end point* (chiamato *tesi*).

2.2 Definizione

Gli algoritmi euristici sono procedure formali dove la soluzione non è garantita essere quella **corretta**. Questo potrebbe sembrare inutile, ma al contrario potrebbe essere utile per diversi motivi:

1. **Costa** molto meno di un algoritmo corretto, in termini di *spazio* e *tempo*.
2. Frequentemente **restituisce** qualcosa di vicino alla soluzione corretta.

3 CLASSIFICAZIONE DEI PROBLEMI

Per definire la *vicinanza* della soluzione, lo **spazio delle soluzioni** sarà dotato di una *metrica* per esprimere una *distanza soddisfacente* della soluzione corrente dalla soluzione corretta. Inoltre sarà dotato di una **distribuzione probabilistica** per esprimere la frequenza soddisfacente delle soluzioni che si trovano ad una distanza soddisfacente dalla soluzione corretta (ovvero esprime quanto spesso l'algoritmo restituisce una soluzione soddisfacente).

2.3 Motivazioni per gli algoritmi euristici

Le euristiche sono la costruzione sia delle **dimostrazioni** che degli **algoritmi**, in caso di successo l'euristica viene abbandonata e la *dimostrazione* viene preservata. Altrimenti, una buona euristica solitamente porta ad un buon risultato, seppur non perfetto.

3 Classificazione dei problemi

Questo corso è incentrato su gli algoritmi euristici applicati ai problemi di **ottimizzazione combinatoria** che sono **basati su soluzioni** (contrapposti a quelli **basati sui modelli**).

Un problema è una domanda che viene effettuata su un **sistema matematico**, la tipologia dei problemi viene classificata in base alla natura delle loro soluzioni.

1. **problemi di decisione:** la loro soluzione è un booleano.
2. **problemi di ricerca:** la loro soluzione è un qualsiasi sottoinsieme *fattibile*.
3. **problemi di ottimizzazione:** la loro soluzione è un numero il quale è il *minimo* o *massimo* valore di una **funzione obiettivo** definita su dei sottointersiemi fattibili.
4. **problemi di conteggio:** la loro soluzione è il *numero* di sottointersiemi fattibili.
5. **problemi di enumerazione:** la loro soluzione è la collezione di tutti i sottointersiemi fattibili.

I problemi di ottimizzazione possono essere combinati con i problemi di ricerca, noi ci concentreremo su questo tipo di problemi, ovvero siamo alla ricerca del **valore ottimale** e del **sottosistema che assume** quel valore.

3.1 Problemi di ottimizzazione e ricerca

Un problema di ottimizzazione e ricerca può essere rappresentato con:

$$opt_{x \in X} f(x)$$

3 CLASSIFICAZIONE DEI PROBLEMI

Dove x rappresenta una **sottosistema fattibile** che è una delle soluzioni, la quale soddisfa le condizioni fornite dal problema. Invece, X è lo **spazio delle soluzioni fattibili**. Invece la funzione, si chiama **funzione obiettivo** ed è mappata in questa maniera $f : X \rightarrow R$, il suo compito è quello di *misurare quantitativamente* la qualità di ogni sottosistema (o soluzione). Generalmente, in quanto problema di ottimizzazione, la funzione obiettivo è *massimizzabile* o *minimizzabile*, questo viene denotato con $opt \in \min, \max$.

Il problema consiste nel determinare il **valore ottimale della funzione obiettivo** assieme alla **soluzione ottima** tale che sia un sottoinsieme. Il valore ottimale della funzione obiettivo viene chiamato f^* , ed è il risultato della seguente equazione:

$$f^* = opt_{x \in X} f(x)$$

Ovvero un valore che consiste nel minimo o massimo della funzione obiettivo. Mentre, la soluzione ottima tale che sia un sottoinsieme si chiamerà x^* :

$$x^* \in X^* = \arg \min_{x \in X} f(x) = x^* \in X : f(x^*) = opt_{x \in X} f(x)$$

Ovvero, vogliamo trovare una soluzione ottima nell'intero insieme di tutte le soluzioni ottime, anche se solitamente una è abbastanza, la notazione *arg* sta per l'intero insieme delle soluzioni (ne basta una).

3.2 perché siamo interessati nei problemi di ottimizzazione e ricerca ?

I problemi di ottimizzazione e ricerca sono di forte interesse poiché diversi campi applicativi richiedono oggetti o strutture caratterizzati da valori molto alti o molto bassi rispetto ad una propria funzione di valutazione.

- Bioinformatica
- Social networks
- Machine learning
- Hardware design
- Stima dei parametri
- Finanza

L'**ottimizzazione esatta** è costosa da un punto di vista computazionale e non sempre desiderabile (per questo gli algoritmi euristicci sono favoriti); perciò, solitamente, le funzioni di valutazione sono delle *approssimazioni* di quello che realmente accade. In questo corso assumeremo il punto di vista dell'ottimizzazione, cercando di ottimizzare al meglio possibile la funzione obiettivo.

Diversi problemi possono spesso essere ridotti in problemi di ottimizzazione e ricerca

3 CLASSIFICAZIONE DEI PROBLEMI

- *Problemi di ricerca* possono essere ridotti rilassando le condizioni da soddisfare, in maniera da allargare la **regione di fattibilità** da X a $X' \supset X$ ed ottenere un problema di ricerca semplice. Si introduce una funzione $d(x)$ per quantificare la distanza di ogni soluzione $x \in X'$ da X . Infine, minimizzando $d(x)$ per trovare x^* tale che $d(x^*) = 0 \Leftrightarrow x^* \in X$.
- Alcuni *problemi di decisione* riguardano l'esistenza di sottosistemi fattibili, e sono identici ai problemi di ricerca (trovare il sottosistema dimostra la sua esistenza).
- Alcuni *problemi di numerazione* riguardano la ricerca di sottosistemi con "buoni" valori di funzioni oggettive in conflitto e permettono *adattamenti diretti* ad algoritmi di ottimizzazione e ricerca.

Tali riduzioni sono spesso possibili ed utili, ma non sempre.

3.3 Problema di ottimizzazione combinatoria

Un problema è un problema CO (*Combinatorial Optimization*) quando la regione di fattibilità X è un insieme finito, e quindi, che abbia un numero finito di *soluzioni fattibili*. Questa sembra un'assunzione molto restrittiva, ma sono presenti molti problemi che hanno un numero infinito di soluzioni che possono essere ridotti a problemi che hanno un numero finito di soluzioni.

Per esempio:

- I problemi infiniti nel discreto possono avere un insieme finito di soluzioni interessanti.
- Alcuni problemi continui possono essere ridotti a problemi di ottimizzazione combinatoria: *Programmazione lineare, Flusso massimo, Costo minimo di flusso, ...*.
- I problemi continui possono essere ridotti in discreto utilizzando il campionamento (solitamente non è molto efficace).
- Le idee concepite per i problemi CO possono essere estese ad altri problemi.

3.4 Euristiche basate sui modelli

In questo corso parleremo solamente di **euristiche basate su soluzioni**, ma è importante conoscere anche la controparte. L'euristiche basate sui modelli descrivono la regione di fattibilità X con un "modello", un esempio tipico è una funzione matematica.

$$opt_{x \in X} f(x) \rightarrow \min_{g_i(\xi) \leq 0} \phi(\xi) \text{ per } i = 1, \dots, m$$

Dove $\xi \in \mathbb{R}^n$, il quale è il vettore delle soluzioni di n numeri reali. Mentre $X = \xi \in \mathbb{R}^n : g_i(\xi) \leq 0, i = 1, \dots, m$, ovvero che la regione di fattibilità è l'insieme dei vettori che soddisfanno tutte le diseguaglianze. L'euristiche basate su

3 CLASSIFICAZIONE DEI PROBLEMI

modelli estrapolano l'informazione derivata dal modello, che sono le proprietà analitiche della funzione ϕ e g_i per $i = 1, \dots, m$.

3.5 Definizione alternativa di CO

Una problema è un problema CO() quando:

1. Il numero di soluzioni fattibili è finito (prima definizione).
2. La regione di fattibilità è $X \subseteq 2^B$ per un dato **ground set** B , ovvero, le *soluzioni fattibili* sono tutte sottoinsiemi del ground set che soddisfa le condizioni adeguate.

Entrambe le definizioni sono equivalenti:

- 2 \implies 1: se il ground set B è finito, ogni collezione $X \subseteq 2^B$ è finita.
- 1 \implies 2: se il numero di soluzioni fattibili è finito, definire B come il loro sovrainsieme ed X la *regione fattibile* sarà la collezione dei singoli elementi di B (una "soluzione" è un insieme contenente una singola soluzione).

In generale, la definizione sofisticata permette un analisi più profonda, perché X non viene semplicemente numerato e viene definito in una maniera *compatta* e *significativa*.

3.6 Euristiche basate su soluzioni

L'euristiche basate su soluzioni considerano le soluzioni come sottoinsiemi del ground set, esse possono essere classificate in:

1. **Euristiche costruttive/distrettive**, iniziano da un sottoinsieme estremamente semplice (può essere \emptyset o B), poi, esse aggiungono/rimuovono gli elementi fino a che non ottengono la soluzione desiderata.
2. **Euristiche di scambio**, iniziano da un sottoinsieme ottenuto in una qualsiasi maniera, poi, scambiano gli elementi fino a che non ottengono la soluzione desiderata.
3. **Euristiche di ricombinazione**, iniziano da una popolazione di sottoinsiemi ottenuta in una qualsiasi maniera, poi, ricombinando differenti sottoinsiemi produrranno una *nuova* popolazione.

I progettisti delle heuristiche possono combinare in maniera creativa gli elementi delle diverse classi delle heuristiche.

3 CLASSIFICAZIONE DEI PROBLEMI

3.6.1 Randomizzazione e memoria

Sono presenti due cose importanti che intervengono nella progettazione di un algoritmo:

- **Randomizzazione**
- **Memoria**

Puoi avere algoritmi che usano o non usano la randomizzazione o la memoria. Questi due elementi sono ortogonali (indipendenti) rispetto alla classificazione delle euristiche basate sulle soluzioni, per ognuna di esse possiamo dire che abbiamo quattro sottoclassi.

1. Utilizzo della randomizzazione:

- **Euristiche puramente deterministiche**
- **Euristiche "randomizzate"**, essenzialmente sono algoritmi che utilizzano come input numeri pseudo-casuali.

2. Utilizzo della memoria:

- Euristiche dove l'input include solamente i **dati del problema**.
- Euristiche dove l'input include anche **soluzioni precedentemente generate**.

Comunemente si utilizza il termine *metaeuristiche* (dal Greco, "oltre le euristiche") per descrivere gli algoritmi euristici che vanno utilizzano la randomizzazione e/o la memoria.

3.7 Rischi da cui stare attenti

1. **Attitudine reverenziale o alla tendenza**, ovvero, nello scegliere un algoritmo basato sul contesto sociale, anziché sul problema.
2. **Attitudine magica**, ovvero, *credere* in un metodo sulla base di un'analogia con un fenomeno fisico e naturale.
3. **Integralismo euristico**, ovvero, utilizzare un'euforia per un problema che ammette l'utilizzo di un'algoritmo esatto.
4. **Sgranocchiare numeri**, ovvero, eseguire sofisticati e complessi calcoli con numeri inaffidabili.
5. **Attitudine SUV**, ovvero, affidarsi alla potenza dell'hardware.
6. **Complicare ulteriormente**, ovvero, introdurre componenti e parametri *ridondanti*, per cercare (fallendo) di migliorare il risultato.
7. **Overfitting**, ovvero, adattare i componenti ed i parametri dell'algoritmo ad un dataset specifico utilizzato nella valutazione sperimentale.

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Inoltre è fondamentale:

- Liberarsi dai pregiudizi.
- Valutare le prestazioni dell'algoritmo in una maniera scientifica.
- Distinguere il contributo di ogni componente dell'algoritmo.
- Implementare efficientemente ogni componente dell'algoritmo.

4 Problemi di ottimizzazione combinatoria

Il ground set è la base sul quale si costruisce l'algoritmo, abbiamo visto che sono presenti molteplici possibilità con le euristiche basate sulle soluzioni, la loro classe cambierà in base al ground set utilizzato. Quindi per prima cosa dobbiamo capire che cosa è (*tipologia*) il **ground set**.

Visiteremo inizialmente un certo numero di problemi, questo sarà utile perché:

- Le idee astratte devono essere applicate concretamente su diversi algoritmi per diversi problemi.
- La stessa idea può avere differente efficacia su diversi problemi.
- Alcune idee funzionano solamente su problemi con una specifica struttura.
- Diversi problemi potrebbero non avere un apparente relazione, cosa che può essere sfruttata per progettare algoritmi.

Una buona conoscenza di diversi problemi ci insegna ad applicare le idee astratte a nuovi problemi e ci insegna come trovare e sfruttare le relazioni tra problemi conosciuti e nuovi.

4.1 Insieme dei problemi pesati

4.1.1 Knapsack Problem (KP)

Il problema dello zaino, *Knapsack Problem*. Il problema consiste nell'avere a disposizione uno zaino che ha una *capacità limitata* ed un insieme di oggetti con differenti *volumi* e *valori*, si vuole riempire lo zaino con gli oggetti di valore massimo (ovviamente non si può mettere dentro tutti gli oggetti).

Dati:

- Insieme elementare E di oggetti.
- Una funzione $v : E \rightarrow \mathbb{N}$ che descrive il **volume** di ogni oggetto.
- Un numero $V \in \mathbb{N}$ che descrive la **capacità** dello zaino.

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

- Una funzione $\phi : E \rightarrow \mathbb{N}$ che descrive il **valore** di ogni oggetto.

Banalmente, il **ground set** è l'insieme degli oggetti $B \equiv E$. La **regione di fattibilità** include tutti i sottoinsiemi degli oggetti il cui volume totale non eccede la capacità V dello zaino.

$$X = \left\{ x \subseteq B : \sum_{j \in x} v_j \leq V \right\}$$

L'obiettivo è quello di massimizzare il valore totale degli oggetti scelti:

$$\max_{x \in X} f(x) = \sum_{j \in x} \phi_j$$

Per esempio, nella seguente tabella sono mostrati tutti gli elementi di E con i relativi valori e volumi. Sapendo che lo zaino ha una capacità massima $V = 8$, consideriamo due **soluzioni candidate**:

E	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1

Figura 4.1: Dataset

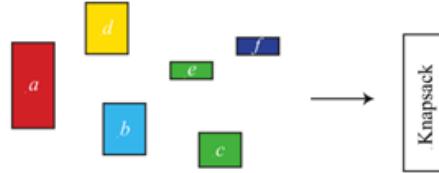


Figura 4.2: Raffigurazione problema dello zaino

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA



$$x' = \{c, d, e\} \in X \quad f(x') = 13$$

$$x'' = \{a, c, d\} \notin X \quad f(x'') = 16$$

Figura 4.3: Soluzioni candidate

- La prima soluzione candidata considera tre elementi la cui somma è $\leq V$, questo significa che la soluzione è un sottoinsieme preso dalla regione di fattibilità X ed è una **soluzione fattibile**.
- La seconda soluzione candidata ha una somma di elementi pari a 10, la quale è $> V$. Quindi questo sottoinsieme non è appartenente alla regione di fattibilità X (ma solo al ground set), perciò verrà chiamata **soluzione infattibile**.

Tra le soluzioni fattibili proposte la **funzione obiettivo** propone di prendere la soluzione *massima*, ma visto che x'' non è fattibile, prenderemo come soluzione x' .

4.1.2 Maximum Diversity Problem (MDP)

Il problema della diversità massima, *Maximum Diversity Problem*, è un problema importante per il corso e verrà utilizzato come esempio per la parte di laboratorio. Questo è un problema definito su uno spazio metrico, quindi uno spazio con la nozione di *distanza*.

Dati:

- Un insieme di punti P .
- Una funzione $d : P \times P \rightarrow \mathbb{N}$, la quale provvede la distanza tra le coppie di punti.
- Un numero $k \in 1, \dots, |P|$, il quale è il numero di punti che si vuole selezionare.

Il problema chiede di selezionare da un insieme di punti P un sottoinsieme di k punti la cui sommatoria delle distanze tra le coppie dei punti sia massima. Questo è un problema CO, perché il numero di sottoinsiemi possibili è finito, ed in particolare è un problema CO perché le soluzioni sono sottoinsiemi del

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

ground set. Il ground set, banalmente, è l'insieme dei punti $B \equiv P$, mentre la regione di fattibilità include tutti i sottoinsiemi composti da k punti.

$$X = \{x \subseteq B : |x| = k\}$$

La funzione obiettivo è la sommatoria di tutte le distanze tra le coppie di punti in x :

$$\max f(x) = \sum_{\substack{x \in X \\ (i,j): i,j \in x}} d_{ij}$$

Per esempio, consideriamo un dataset costituito da 7 punti e considerando un $k = 3$, questo significa che vogliamo trovare un sottoinsieme costituito da 3 punti tale che le coppie abbiano distanza massima.

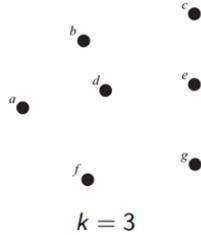


Figura 4.4: Dataset

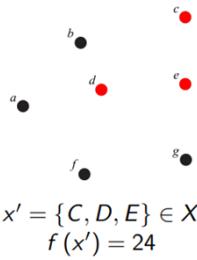


Figura 4.5: Prima soluzione candidata del MDP

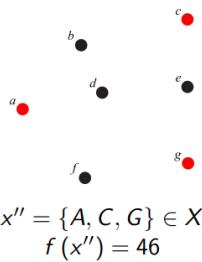


Figura 4.6: Seconda soluzione candidata del MDP

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

La prima soluzione x' , considerando una metrica *non fornita* ha come valutazione della funzione obiettivo $f(x') = 24$, ed è un sottoinsieme tale per cui la sua cardinalità sia $\leq k$, e quindi appartenente alla regione di fattibilità X .

La seconda soluzione ha come soluzione della funzione obiettivo $f(x'') = 46$, questo è anche fattibile visto che $|x''| = k$, ed è una soluzione preferibile alla prima visto che stiamo cercando il sottoinsieme che soddisfi massimo della funzione obiettivo.

4.1.3 Interludio 1: la funzione obiettivo

Fermiamoci un attimo e pensiamo qualcosa a riguardo di questo problema, in particolare soffermiamoci sulla funzione obiettivo. Questa viene data come funzione che ha come dominio la regione di fattibilità e giunge al codominio all'insieme dei numeri naturali.

$$f : X \rightarrow \mathbb{N}$$

Il calcolo di questa funzione potrebbe essere molto complesso ed estenuante, ogni singola soluzione ha il proprio valore rispetto alla funzione obiettivo e si dovrebbe andare a controllare ogni volta in una tabella per svolgere il calcolo, non è una cosa molto interessante da fare.

Questo ovviamente, non è il caso dei precedenti problemi in quanto semplici da svolgere.

In particolare il problema KP una funzione obiettivo **additiva**, nel calcolare la f dobbiamo effettuare delle somme con il valore della funzione ausiliaria ϕ definita sul ground set, ricordando che nel problema KP, $B \equiv E$.

$$\phi : B \rightarrow \mathbb{N} \text{ induce } f(x) = \sum_{j \in x} \phi_j : X \rightarrow \mathbb{N}$$

Questo è interessante perché significa che si deve memorizzare solamente il valore della funzione ausiliaria ϕ , i quali sono $|B|$ valori, e non 2^B , come sarebbe per i valori della funzione obiettivo che è definita su X .

Lo stesso accade per il problema MDP e la sua funzione ausiliaria d , anche se quest'ultima ha una funzione obiettivo quadratica, poi fornito un $n = |B|$ di punti nel caso peggiore si dovranno sommare $\frac{n(n-1)}{2}$ (numero di archi in un grafo completo) distanze, tuttavia però il calcolo rimane "solamente" una somma, quindi avente una **complessità polinomiale**.

Nel problema KP una volta che il valore viene fornito per uno specifico sottoinsieme, rimane possibile modificare gli elementi e ricalcolare il valore della funzione obiettivo facilmente. Il valore della funzione obiettivo nel caso del MDP va trattato in maniera differente per essere calcolato in tempo lineare (poiché quadratica).

Un'altra importante osservazione è che il problema KP e MDP sono definiti sull'intero insieme delle possibili (non fattibili) soluzioni 2^B , e questo è generalmente inutile visto che stiamo cercando una soluzione fattibile (in alcuni casi questo però sarà utile).

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Per riassumere, quando guardiamo un problema cerchiamo di capire come la funzione obiettivo è costituita:

- È una funzione additiva?
- È una funzione quadratica?
- È una funzione semplice da calcolare?
- È una funzione semplice da aggiornare?
- Su cosa è definita la funzione obiettivo?

4.2 Insieme dei problemi di partizionamento

In questi problemi un insieme di oggetti viene fornito, l'obiettivo consiste nel dividerlo in sottoinsiemi ottenendo una partizione con alcune peculiarità.

4.2.1 Bin Packing Problem (BPP)

Il Bin Packing Problem (BPP), si ha un insieme di oggetti con un *volume*, e si vuole mettere questi oggetti all'interno di container con una *capacità fissa* (fornita) utilizzando il **minimo numero di container**.

Dati:

- Insieme E di oggetti.
- Una funzione $v : E \rightarrow \mathbb{N}$ che fornisce il volume per un dato oggetto $e \in E$.
- Un insieme C di containers.
- Un numero $V \in \mathbb{N}$ il quale rappresenta la capienza massima dei container (volume massimo contenibile).

La prima domanda che ci si vuole porre è: *è un problema di ottimizzazione combinatoria?*

Il ground set è definito come $B = E \times C$, dove ogni elemento di B è definito da una coppia $\langle \text{oggetto}, \text{container} \rangle$. Una soluzione per questo problema è un sottoinsieme formato da oggetti di questo tipo, il prodotto cartesiano è necessario poiché si deve selezionare un oggetto ed inserirlo in un determinato container.

Una volta che la lista di coppie contenenti gli elementi di E è costruita, una soluzione candidata sarà ottenuta (un sottoinsieme del ground set, ma a noi questo non basta, voglia che sia un sottoinsieme della regione di fattibilità).

Consideriamo B_e come il sottoinsieme del ground set dove gli oggetti delle coppie provengono da E (i container sono tutti i possibili), e B_c come il sottoinsieme del ground set dove i container nelle coppie degli elementi provengono da C (gli elementi sono tutti i possibili).

$$B_e = \{(i, j) \in B : i = e\}$$

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

$$B_c = \{(i, j) \in B : i = c\}$$

La **regione di fattibilità** include tutte le partizioni degli oggetti tra i container tale per cui non ecceda la capacità di un qualsiasi container.

$$X = \left\{ x \subseteq B : |x \cap B_e| = 1 \forall e \in E, \sum_{(e,c) \in B^c} v(e) \leq V \forall c \in C \right\}$$

La prima parte dell'espressione è un vincolo sul sottoinsieme delle soluzioni fattibili x , dice che l'intersezione tra il sottoinsieme delle soluzioni fattibili ed il ground set deve avere modulo 1. Questo significa che gli elementi all'interno di x devono essere presenti solamente una volta rispetto a gli elementi E del ground set, ovvero $\in B_e$.

La seconda parte dell'espressione anch'essa è un vincolo ma rispetto al volume massimo dei container, dice che la somma dei volumi di ogni singolo elemento del sottoinsieme fattibile non deve eccedere la capacità massima dei container V .

L'obiettivo è quello di minimizzare il numero di container utilizzati:

$$\min f(x) = \min_{x \in X} |c \in C : x \cap B^c \neq \emptyset|$$

Per esempio, consideriamo degli oggetti con diversi volumi e sia data una capacità massima dei container pari a $V = 4$.

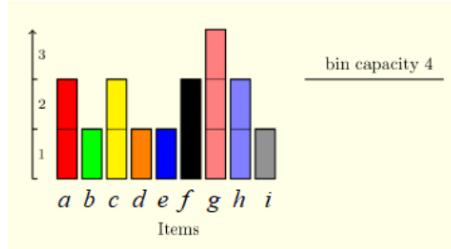


Figura 4.7: Dataset

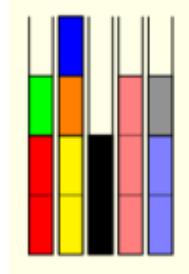


Figura 4.8: Prima soluzione candidata del Bin Packing Problem

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Consideriamo la prima soluzione proposta, visto che la lista dei prodotti cartesiano rispetta i due vincoli nella definizione della regione di fattibilità la soluzione x' è una **soluzione fattibile**.

$$x' = \{(a, 1), (b, 1), (c, 2), (d, 2), (e, 2), (f, 3), (g, 4), (h, 5), (i, 5)\} \in X$$

$$f(x') = 5$$

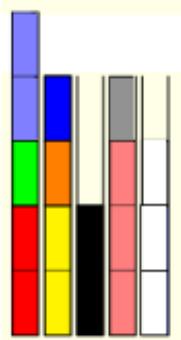


Figura 4.9: Seconda soluzione candidata del Bin Packing Problem

Invece, la seconda soluzione proposta è una **soluzione infattibile**, questo perché gli elementi non stanno rispettando il secondo vincolo sul volume, e quindi questo sottoinsieme non è compreso all'interno della regione di fattibilità.

$$x'' = \{(a, 1), (b, 1), (c, 2), (d, 2), (e, 2), (f, 3), (g, 4), (h, 1), (i, 4)\} \notin X$$

$$f(x'') = 4$$

Considerando il caso in cui x'' fosse una soluzione fattibile avremmo scelto quella tra le soluzioni proposte, poiché la funzione obiettivo effettua l'ottimizzazione sul minimo.

4.2.2 Parallel Machine Scheduling Problem (PMSP)

Il Parallel Machine Scheduling Problem (PMSP), è un problema nel quale un insieme di attività (tasks) deve essere diviso lungo un set di macchine in modo che il *tempo di completamento* sia minimizzato.

Dati:

- Un insieme T di tasks (o *attività*).
- Una funzione $d : T \rightarrow \mathbb{N}$ che descrive la lunghezza (temporale) di ogni task.
- Un insieme di M macchine.

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Come prima, il ground set è dato dal prodotto cartesiano di due set forniti:

$$B \equiv T \times M$$

Significa che la soluzione deve essere una coppia

$$\langle task, macchina \rangle$$

. È importante sottolineare che la sequenza in cui i task sono eseguiti non è rilevante, invece è rilevante il **tempo di completamento**, ovvero il tempo con cui l'ultimo task termina (o il tempo in cui una macchina completa l'esecuzione dei suoi tasks).

La regione di fattibilità include tutte le partizioni delle attività nella macchine:

$$X = \{x \subseteq B : |x \cap B_t| = 1 \forall t \in T\}$$

La **funzione obiettivo** ha come obiettivo quello di minimizzare il massimo della sommatoria delle lunghezze di tempo per ogni task di ogni macchina:

$$\min f(x) = \max_{x \in X} \sum_{m \in M} \sum_{t:(t,m) \in x} d_t$$

In parole povere, vogliamo trovare il sottoinsieme x che minimizza il tempo di completamento di ciascuna macchina, dove il tempo di completamento per un singolo task è $\sum_{t:(t,m) \in x} d_t$.

Per esempio, consideriamo il seguente insieme di dati per tre macchine, $|M| = 3$, e sette task differenti $|T| = 7$.

$T = \{T1, T2, T3, T4, T5, T6\}$														
$M = \{M1, M2, M3\}$														
<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th style="text-align: left;">task</th> <th style="text-align: center;">$T1$</th> <th style="text-align: center;">$T2$</th> <th style="text-align: center;">$T3$</th> <th style="text-align: center;">$T4$</th> <th style="text-align: center;">$T5$</th> <th style="text-align: center;">$T6$</th> </tr> </thead> <tbody> <tr> <td style="text-align: left;">d</td> <td style="text-align: center;">80</td> <td style="text-align: center;">40</td> <td style="text-align: center;">20</td> <td style="text-align: center;">30</td> <td style="text-align: center;">15</td> <td style="text-align: center;">80</td> </tr> </tbody> </table>	task	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$	d	80	40	20	30	15	80
task	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$								
d	80	40	20	30	15	80								

Figura 4.10: Dataset del Parallel Machine Scheduling Problem

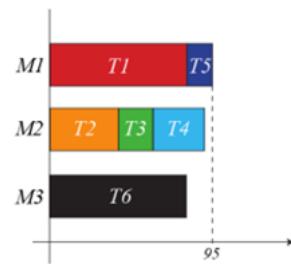


Figura 4.11: Prima soluzione Parallel Machine Scheduling Problem

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Consideriamo la prima soluzione proposta:

$$x' = \{(T1, M1), (T2, M2), (T3, M2), (T4, M2), (T5, M1), (T6, M3)\} \in X$$

$$f(x') = 95$$

Possiamo notare che questa è una **soluzione fattibile**, visto che ogni task accade al meno ed al massimo una volta. Notiamo che il valore assunto dalla funzione obiettivo è 95, questo proprio perché l'ultimo task ha un tempo di completamento pari a 95.

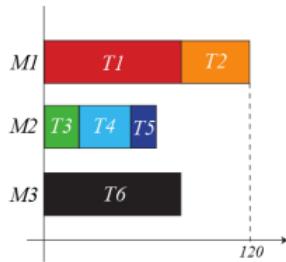


Figura 4.12: Seconda soluzione Parallel Machine Scheduling Problem

Consideriamo la seconda soluzione proposta:

$$x'' = \{(T1, M1), (T2, M1), (T3, M2), (T4, M2), (T5, M2), (T6, M3)\} \in X$$

$$f(x'') = 120$$

Notiamo che anche questa è una **soluzione fattibile** visto che ogni task accade al minimo ed al massimo una volta, in questo caso la funzione obiettivo assume come valore 120. Questo significa che fra le due soluzioni proposte la prima è quella **ottima**.

4.2.3 Interludio 2: la funzione obiettivo, ancora

È necessario familiarizzare con il fatto che il ground set B non è sempre uno degli insiemi forniti nel problema, ma può essere formato dalla combinazione (come il prodotto cartesiano) di diversi insiemi forniti. Ora affrontiamo la domande proposte nell'ultimo interludio.

Le funzioni oggettivi di che tipo sono? (additive, quadratiche,...) Questa volta le funzioni oggettive per il BPP e PMSP **non** sono additive, e non sono neanche banali. È presente un algoritmo polinomiale per calcolare la funzione obiettivo, seppur non complesso, non è semplice come per i problemi precedenti.

Notiamo che piccole modifiche alle soluzioni hanno un impatto *variabile* sull'obiettivo, per esempio consideriamo la seconda soluzione x'' del PMSP:

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

- Spostare il task T_5 sulla macchina M_1 , allunga il tempo di completamento complessivo M_1 , il risultato della funzione obiettivo cambia perché viene incrementato del task spostato (*impatto corrispondente al tempo del task spostato*).
- Spostare il task T_5 sulla macchina M_3 , non modifica il tempo di completamento complessivo delle macchine, il risultato della funzione obiettivo rimane lo stesso (*impatto zero*).
- Spostare il task T_2 sulla macchina M_2 , comporta una modifica dei tempi di completamento complessivi, il risultato della funzione obiettivo cambia poiché l'ultimo task viene spostato (*impatto intermedio*)

In fatti, l'impatto di una modifica di una soluzione dipende :

- Da entrambi gli elementi modificati.
- E dagli elementi non modificati (questo è contrario alle cose dette nell'interludio 1).

Un punto interessante è che la funzione obiettivo del PMSP tende ad essere **piatta**, ovvero che sono presenti molteplici soluzioni all'interno del problema dove il valore della funzione rimane lo stesso anche se avvengono delle modifiche (l'esempio precedente, la soluzione x'' rimane fissa per diverse combinazioni su 120).

4.3 Problemi delle funzioni logiche

4.3.1 The Max-SAT problem

Il problema del Max-Sat, sia da una formula in **forma normale congiuntiva** (CNF, *Conjunctive Normal Form*), si vogliono fornire in ingresso dei valori di verità alle variabili logiche della CNF tali per cui la formula venga soddisfatta (valutata come vera).

Dati:

- Un insieme V di **variabili logiche** x_j con valori in $\mathbb{B} \in \{0, 1\}$.
- Un **letterale** $l_j(x) \in x_j, \bar{x}_j$ che è una funzione che consiste in una variabile logica *affermata* o *negata*.
- Una **formula logica** $C_i(x) = l_{i,1} \vee \dots \vee l_{i,n_i}$, la quale è una disgiunzione o *somma logica* (OR) di letterali. Soddisfare una formula logica significa fargli assumere valore 1.
- Una formula in **forma normale congiuntiva** $CNF(x) = C_1 \wedge \dots \wedge C_n$ è una congiunzione di *prodotti logici* di formule logiche.
- Una funzione w che provvede dei *pesi* per la formula CNF. La funzione associa ogni formula logica della CNF ad un rispettivo peso.

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Visto che la soluzione consiste in un sottoinsieme caratterizzato dall'assegnamento di valori di verità a variabili logiche, il **ground set** sarà il prodotto cartesiano fra le variabili logiche e l'insieme dei numeri booleani:

$$B = V \times \mathbb{B} = \{(x_1, 0), (x_1, 1), \dots, (x_n, 0), (x_n, 1)\}$$

La **regione di fattibilità** è l'insieme delle soluzioni fattibili tali che una *variabile* venga considerata al più una volta. Essa include tutti sottoinsiemi costituenti gli assegnamenti semplici che sono:

- **completi**, ovvero che ad ogni variabile corrisponde *almeno* un letterale.
- **consistenti**, ovvero che per ogni variabile corrisponde *al massimo* un letterale.

$$X = \{x \subseteq B : |x \cap B_v| = 1 \forall v \in V\}$$

$$B_{xj} = \{(x_j, 0), (x_j, 1)\}$$

La **funzione obiettivo** (come sempre ottimizzata):

$$\max f(x) = \sum_{x \in X} w_i$$

L'obiettivo è quello di massimizzare il peso totale della *formula logica soddisfatta* segnata come $C_i(x) = 1$ per $i = 1, \dots, n$ (dove n è il numero di formule logiche presenti).

Consideriamo il seguente esempio:

$$\begin{aligned} V &= \{x_1, x_2, x_3, x_4\} \\ L &= \{x_1, \bar{x}_1, x_2, \bar{x}_3, x_3, \bar{x}_4, x_4\} \\ C_1 &= \bar{x}_1 \vee x_2 \dots C_7 = x_2 \\ CNF &= (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_4) \wedge (\bar{x}_2 \vee \bar{x}_4) \wedge x_1 \wedge x_2 \\ w_i &= 1 \forall C_i \end{aligned}$$

Consideriamo adesso la seguente soluzione:

$$x = \{(x_1, 0), (x_2, 0), (x_3, 1), (x_4, 1)\}$$

La funzione obiettivo per questa soluzione assume valore $f(x) = 5$, significa che soddisfa 5 formule delle 7.

Risulta semplice trovare il valore della funzione obiettivo? Non proprio, la complessità della funzione oggettive è polinomiale

In caso di pesi uniformi sono presenti un campo ristretto di valori, che vanno da 0 a n , il numero di formule logiche, anche se sono presenti $2^{|V|}$ combinazioni che possono essere considerate.

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

4.4 Problemi con matrici numeriche

4.4.1 Set covering problem (SCP)

Data una **matrice binaria** ed una **funzione costo** definita per ogni colonna della matrice (come vettore), si vuole selezionare il sottoinsieme di colonne che coprono tutte le righe di costo minimo.

Dati:

- Matrice binaria $A \in \mathbb{B}^{m,n}$ con insieme delle righe R e insieme delle colonne C .
- La colonna $j \in C$ copre la riga $i \in R$ quando $a_{ij} = 1$.
- Una funzione $c : C \rightarrow \mathbb{N}$ provvede il costo di ogni colonna.

Il **ground set** è l'insieme delle colonne.

$$B \equiv C$$

La **regione di fattibilità** include tutti i sottoinsiemi delle colonne che coprono tutte le righe.

$$X = \left\{ x \subseteq B : \sum_{j \in x} a_{ij} \geq 1 \quad \forall i \in R \right\}$$

L'obiettivo è *minimizzare* il costo totale delle colonne selezionate, la **funzione obiettivo** è additiva, molto veloce da calcolare ed aggiornare, però la **fattibilità** non è semplice da ottenere.

$$\min f(x) = \sum_{j \in x} c_j$$

Consideriamo il seguente esempio di una matrice con il relativo vettore dei costi:

c	4 6 10 14 5 6																														
A	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> </table>	0	1	1	1	1	0	0	0	1	1	0	0	1	1	0	0	0	1	0	0	0	1	1	1	1	1	1	0	1	0
0	1	1	1	1	0																										
0	0	1	1	0	0																										
1	1	0	0	0	1																										
0	0	0	1	1	1																										
1	1	1	0	1	0																										

Figura 4.13: Dataset del Set Covering Problem (SCP)

Notiamo che la terza e la quinta riga (iniziano dall'alto) sono coperte dalla prima colonna. Infatti, "set covering", si coprono i *set* (righe) con *subset* (colonne).

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Consideriamo adesso una prima soluzione proposta:

A	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>2</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>3</td></tr> </table>	0	1	1	1	1	0	2	0	0	1	1	0	0	1	1	1	0	0	0	1	1	0	0	0	1	1	1	1	1	1	1	0	1	0	3
0	1	1	1	1	0	2																														
0	0	1	1	0	0	1																														
1	1	0	0	0	1	1																														
0	0	0	1	1	1	1																														
1	1	1	0	1	0	3																														

Figura 4.14: Prima soluzione proposta del SCP

$$\begin{aligned}x' &= c_1, c_2, c_5 \in X \\f(x') &= 19\end{aligned}$$

La prima soluzione x' è una **soluzione fattibile**, perché ogni riga ha almeno un elemento $a_{ij} \geq 1$.

A	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>2</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>2</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>2</td></tr> </table>	0	1	1	1	1	0	1	0	0	1	1	0	0	0	1	1	0	0	0	1	2	0	0	0	1	1	1	2	1	1	1	0	1	0	2
0	1	1	1	1	0	1																														
0	0	1	1	0	0	0																														
1	1	0	0	0	1	2																														
0	0	0	1	1	1	2																														
1	1	1	0	1	0	2																														

Figura 4.15: Seconda soluzione proposta del SCP

$$\begin{aligned}x'' &= c_1, c_5, c_6 \notin X \\f(x'') &= 15\end{aligned}$$

La seconda soluzione x'' **non è una soluzione fattibile**, dato che la Seconda riga non è coperta da almeno un elemento delle colonne. Comunque, nel caso in cui la seconda soluzione sia una soluzione fattibile, allora quest'ultima sarebbe una soluzione migliore della prima.

4.4.2 Interludio 3: il test di fattibilità

Solitamente, gli algoritmi euristici **richiedono** di risolvere il seguente problema: *Dato un sottoinsieme x , è fattibile o infattibile?*, in breve $x \in X?$. Come risolvere questo problema? Innanzitutto, è un **problema di decisione**.

Consideriamo SCP, la fattibilità può essere decisa passando per ogni riga e sommando gli 1 che appaiono nella colonna selezionata: se una qualsiasi riga ha una somma complessiva pari a 0 la soluzione sarà **infattibile**.

Nel caso del KP, il test di fattibilità richiede di calcolare dalla soluzione e testare un singolo numero (il peso totale rispetto alla capacità dello zaino) proprio come nel MDP, dove la cardinalità della soluzione si trovava sotto una restrizione k .

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Altri problemi come il Max-SAT ed il PMSP richiedono di testare la fattibilità su un singolo insieme di numeri (numero di variabili logiche non si ripete nella soluzione $x \cap B_v$), mentre problemi come il BPP richiedono di testare su diversi insiemi di numeri (i volumi degli oggetti rispetto alla capacità dei container).

Alcune modifiche alle soluzioni vengono vietate *a priori* per evitare l'inammissibilità delle soluzioni. Supponiamo di avere una soluzione fattibile per il MDP, una qualsiasi modifica in cui il numero di punti non è uguale al numero di punti aggiunti rende la soluzione **infattibile**.

Alcune modifiche non garantiscono **inammissibilità** (unfeasible) della soluzione, le quali richiedono un test *a posteriori* come nel PMSP.

4.5 Set Packing Problem

Il Set Packing Problem è un problema molto simile al precedente SCP, questo perché appartiene alla stessa classe di problemi e provvede un valore per ogni colonna.

Dati:

- Una **matrice binaria** $A \in \mathbb{B}^{m,n}$ con insieme delle righe R e insieme delle colonne C .
- Sia definito un conflitto tra due colonne $j', j'' \in C$ quando $a_{ij'} = a_{ij''} = 1$.
- Una funzione $\phi : C \rightarrow \mathbb{N}$ che provvede il valore di ogni colonna.

Il **ground set** è ancora l'insieme delle colonne:

$$B \equiv C$$

La **regione di fattibilità** include tutti i sottoinsieme di colonne che non sono in conflitto:

$$X = \left\{ x \subseteq B : \sum_{j \in x} a_{ij} \leq 1 \forall i \in R \right\}$$

L'obiettivo consiste nello scegliere le colonne di valore massimo senza che siano presenti *conflicti*.

$$\max_{x \in X} f(x) = \sum_{j \in x} \phi_j$$

Prendiamo in considerazione la seguente matrice:

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

φ	4	6	10	14	5	6
A	0	1	0	0	1	0
	0	0	1	1	0	0
	1	0	0	0	0	1
	0	0	0	1	1	1
	1	1	1	0	0	0

Figura 4.16: Dataset Set Packing Problem

A	0	1	0	0	1	0	1
	0	0	1	1	0	0	1
	1	0	0	0	0	1	1
	0	0	0	1	1	0	1
	1	1	1	0	0	0	1

Figura 4.17: Prima soluzione proposta del SPP (*Set Packing Problem*)

La prima soluzione proposta è $x' = \{c_2, c_4\} \in X$, con valutazione della funzione obiettivo $f(x') = 20$. Questa è una soluzione **fattibile**, visto che non presenti conflitti sulle righe delle colonne selezionate.

A	0	1	0	0	1	0	1
	0	0	1	1	0	0	0
	1	0	0	0	0	1	2
	0	0	0	1	1	0	1
	1	1	1	0	0	0	1

Figura 4.18: Seconda soluzione proposta del SPP (*Set Packing Problem*)

In questo caso **non è una soluzione fattibile**, visto che avviene il conflitto sulla terza riga, anche quando la valutazione della funzione obiettivo era meglio della prima soluzione, le somme valutate non sono ottenute dalla regione di fattibilità X .

4.6 Set Partitioning Problem

Il problema di Set Partitioning (non lo chiameremo con l'acronimo inglese), è una fusione dei due precedenti problemi SCP e SPP.

Dati:

- Una **matrice binaria** $A \in \mathbb{B}^{m,n}$ con insieme delle righe R e insieme delle colonne C .

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

- Una funzione $c : C \rightarrow \mathbb{N}$ che fornisce il costo di ogni colonna.

La risoluzione del problema prevede di selezionare il sottoinsieme di *costo minimo* delle colonne che non sono in conflitto. Il **ground set**, anche questa volta, è l'insieme delle colonne:

$$B \equiv C$$

La **regione di fattibilità** include tutti i sottoinsiemi di colonne che coprono tutte le righe che non sono in conflitto:

$$X = \left\{ x \subseteq C : \sum_{j \in x} a_{ij} = 1 \forall i \in R \right\}$$

$$\min f(x) = \sum_{j \in x} c_j$$

Consideriamo la seguente matrice binaria:

c	4 6 10 14 5 6																														
A	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tbody> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </tbody> </table>	0	1	0	0	1	0	0	0	1	1	0	0	1	0	0	0	0	1	0	0	0	1	1	0	1	1	1	0	0	0
0	1	0	0	1	0																										
0	0	1	1	0	0																										
1	0	0	0	0	1																										
0	0	0	1	1	0																										
1	1	1	0	0	0																										

Figura 4.19: Dataset del Set Partitioning Problem

Ora prendiamo in considerazione la prima soluzione proposta:

A	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tbody> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> </tbody> </table>	0	1	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	0	0	1	1	0	0	0	1	1	0	1	1	1	1	0	0	0	1
0	1	0	0	1	0	1																														
0	0	1	1	0	0	1																														
1	0	0	0	0	1	1																														
0	0	0	1	1	0	1																														
1	1	1	0	0	0	1																														

Figura 4.20: Prima soluzione del SPP (Set Partitioning Problem)

$$x' = \{c_2, c_3, c_6\} \in X$$

$$f(x') = 26$$

Notiamo che la soluzione x' è un **soluzione fattibile**, visto che gli elementi appartenenti alle selezionati si trovano colonne si trovano in una maniera da non generare conflitti lungo le righe (la sommatoria degli elementi lungo le righe non porta ad un risultato diverso da 1).

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Ora consideriamo la seconda soluzione candidata:

A	<table border="1" style="border-collapse: collapse; width: 100%; height: 100%;"> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>2</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table>	0	1	0	0	1	0	1	0	0	1	1	0	0	0	1	0	0	0	0	1	2	0	0	0	1	1	0	1	1	1	1	0	0	0	1
0	1	0	0	1	0	1																														
0	0	1	1	0	0	0																														
1	0	0	0	0	1	2																														
0	0	0	1	1	0	1																														
1	1	1	0	0	0	1																														

Figura 4.21: Seconda soluzione del SPP (*Set Partitioning Problem*)

$$\begin{aligned} x'' &= \{c_1, c_5, c_6\} \notin X \\ f(x'') &= 15 \end{aligned}$$

Anche se il risultato della funzione obiettivo ha un valore migliore (più piccolo) rispetto a quello della prima soluzione, la soluzione x'' **non è una soluzione fattibile**. Questo perché, gli elementi sulla terza riga si trovano in conflitto ($\sum_{j \in x} a_{ij} \geq 1$), ed anche perché la seconda riga non viene *coperta* da alcuna colonna ($\sum_{j \in x} a_{ij} = 0$).

4.7 Interludio 4: ricerca di soluzioni fattibili

Gli algoritmi euristici spesso richiedono di risolvere un altro problema: *trovare una soluzione che fattibile* $x \in X$, questo è un **problema di ricerca**. Chiaramente dato che le soluzioni sono definite da una soluzione iniziale, le euristiche di scambio e ricombinazione hanno bisogno di partire da un sottoinsieme valido tale per cui esso stesso sia una soluzione fattibile.

In base al problema la soluzione può essere banale:

- Alcuni insiemi sono sempre fattibili: $x = \emptyset$ (come nel KP, SPP) o $x = B$ (nel SCP).
- Alcune soluzioni casuali soddisfano un vincolo come $|x| = k$ (nel MDP).
- Alcune soluzioni casuali soddisfano vincoli consistenti, come assegnare un task per ogni macchina come nel PMSP, o un valore ad ogni variabile logica come nel Max-SAT.

Oppure può essere difficile:

- Nel BPP il numero di container deve essere sufficientemente grande.
- Nel SPP non è conosciuto alcun algoritmo polinomiale per risolvere il problema.

Alcuni algoritmi ingrandiscono al regione di fattibilità da X a X' (processo detto "rilassamento"), la funzione obiettivo f deve essere estesa anch'essa da X a X' , ma spesso $X' \setminus X$ provvede delle soluzioni migliori.

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

4.8 Problemi sui grafi

4.8.1 Vertex Cover Problem

Dato un grafo indiretto $G = (V, E)$, selezionare un sottoinsieme di vertici di cardinalità minima tale che ogni arco del grafo sia incidente a quest'ultimo.

Il **ground set** è l'insieme dei vertici:

$$B \equiv V$$

La **regione di fattibilità** include tutti i sottoinsiemi dei vertici tali che gli archi del grafo siano incidenti ad essi:

$$X = \{x \subseteq V : x \cap (i, j) \neq \emptyset \forall (i, j) \in E\}$$

L'obiettivo è minimizzare il numero di vertici selezionati:

$$\min_{x \in X} f(x) = |x|$$

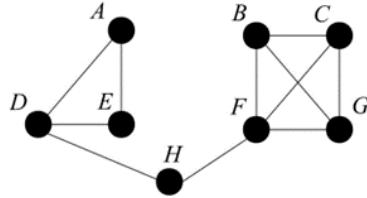


Figura 4.22: Dataset del Vertex Covering Problem

Prima soluzione proposta:

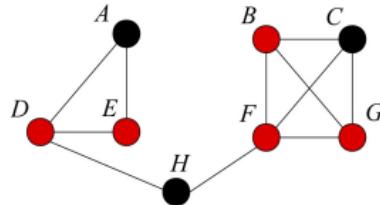


Figura 4.23: Prima soluzione proposta per il VCP

$$\begin{aligned} x' &= \{B, D, E, F, G\} \in X \\ f(x') &= 5 \end{aligned}$$

Notiamo che x' è una **soluzione fattibile**, questo perché il sottoinsieme di vertici selezionato interseca ogni arco del grafo (appartiene alla regione di fattibilità).

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Guardiamo la seconda soluzione proposta:

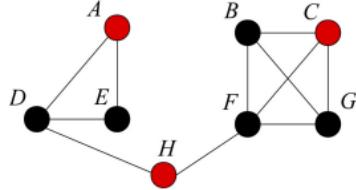


Figura 4.24: Seconda soluzione proposta per il VCP

Invece, la seconda soluzione **non è una soluzione fattibile**, anche se la funzione obiettivo porta ad un risultato che è più convincente della soluzione precedente, questo sottoinsieme non appartiene alla regione di fattibilità (il sottoinsieme di vertici selezionato non è incidente a tutti gli archi del grafo, e.g.: (d, e)).

4.8.2 Maximum Clique Problem (MCP)

Dati:

- Un **grafo indiretto** $G = (V, E)$.
- Una funzione $w : V \rightarrow \mathbb{N}$ che provvede il peso di ogni vertice.

Selezionare il sottoinsieme di coppie di vertici adiacenti di peso massimo. Il **ground set** è l'insieme dei vertici.

$$B \equiv V$$

La **regione di fattibilità** include tutti i sottoinsiemi di coppie di vertici adiacenti.

$$X = \{x \subseteq V : (i, v) \in E \ \forall i \in x, \forall j \in x \setminus \{i\}\}$$

L'obiettivo è quello di massimizzare il peso dei vertici selezionati:

$$f(x) = \sum_{j \in x} w_j$$

Consideriamo il seguente grafo indiretto costituito da **pesi uniformi** $w_i = 1 \forall i \in V$.

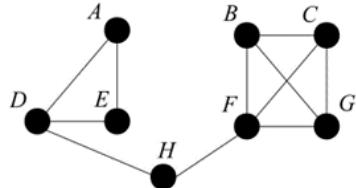


Figura 4.25: Dataset per il Maximum Clique Problem

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Consideriamo la prima soluzione proposta:

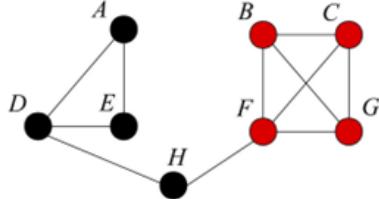


Figura 4.26: Prima soluzione proposta per il MCP

$$x' = \{B, C, F, G\} \in X$$

$$f(x') = 4$$

La prima soluzione proposta è una soluzione **fattibile**, visto che ogni coppia nel sottoinsieme di vertici presenta un arco tra di loro.

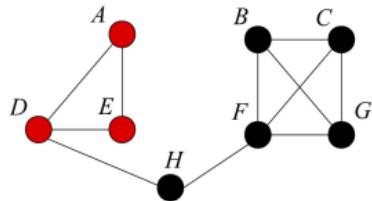


Figura 4.27: Seconda soluzione proposta per il MCP

$$x' = \{A, D, E\} \in X$$

$$f(x') = 3$$

La seconda soluzione proposta è anch'essa una soluzione **fattibile**, per lo stesso motivo precedente.

4.8.3 Maximum Independent Set

Questo problema è opposto al MCP, vogliamo trovare un sottoinsieme di vertici di peso massimo che non è connesso da archi. Dati:

- Un **grafo indiretto** $G = (V, E)$.
- Una funzione $w : V \rightarrow \mathbb{N}$ che provvede un peso per ogni arco.

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Il **ground set** è l'insieme dei vertici.

$$B \equiv V$$

La **regione di fattibilità** include tutti i sottoinsiemi di vertici i cui archi *non sono adiacenti*.

$$X = \{x \subseteq B : (i, j) \notin E \forall i \in x, \forall j \in x \setminus \{i\}\}$$

L'obiettivo è quello di massimizzare il peso dei vertici selezionati.

$$\max f(x) = \sum_{x \in X} w_j$$

Consideriamo il seguente grafo indiretto con *pesi uniformi*.

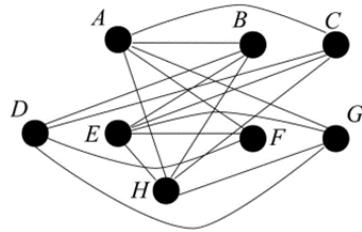


Figura 4.28: Dataset per il Maximum Independent Set

Adesso consideriamo la prima soluzione proposta:

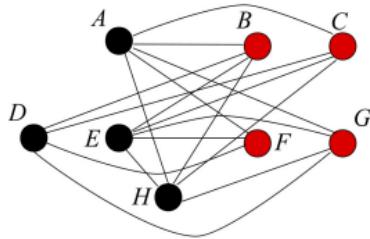


Figura 4.29: Prima soluzione per il MIS

$$x' = \{B, C, F, G\} \in X$$

$$f(x') = 4$$

La soluzione x' è una soluzione **fattibile**, ogni vertice del sottoinsieme proposto *non è connesso* con un altro vertice dello stesso sottoinsieme.

Consideriamo adesso una seconda proposta:

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

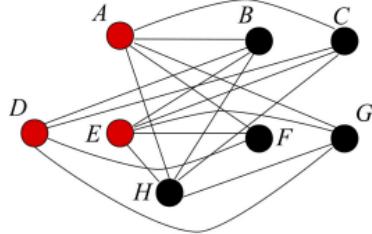


Figura 4.30: Seconda soluzione per il MIS

$$x'' = \{A, D, E\} \in X$$

$$f(x'') = 3$$

Anche la seconda soluzione è una **soluzione fattibile**.

4.8.4 Interludio 5: le relazioni tra i problemi

Come al solito le domande che uno si dovrebbe porre per un qualsiasi problema sono le solite :

- *Come si calcola la funzione obiettivo?*
- *Cosa succede se togliamo un vertice?*
- *Come si verifica la fattibilità?*
- *Cosa succede se aggiungiamo o rimuoviamo un vertice ad una soluzione fattibile?*

Questi ultimi tre problemi sui grafi che abbiamo affrontato erano molto simili. Dovrebbe essere già noto dalla **teoria delle complessità computazionali** che alcuni problemi possono essere **ridotti** ad altri problemi, e che si possa **utilizzare** un problema per risolvere un altro.

Un chiaro esempio è il seguente:

- Si parte dall'istanza iniziale del MCP, con un grafo $G = (V, E)$.
- Si costruisce il **grafo complementare** $\bar{G} = (V, (V \times V) \setminus E)$ (un grafo tale per cui i vertici adiacenti nel grafo originario ora non lo sono più e viceversa).
- Si cerca una soluzione ottima per il MISP su \bar{G} .
- I vertici corrispondenti danno una soluzione ottimale del MCP su G .

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

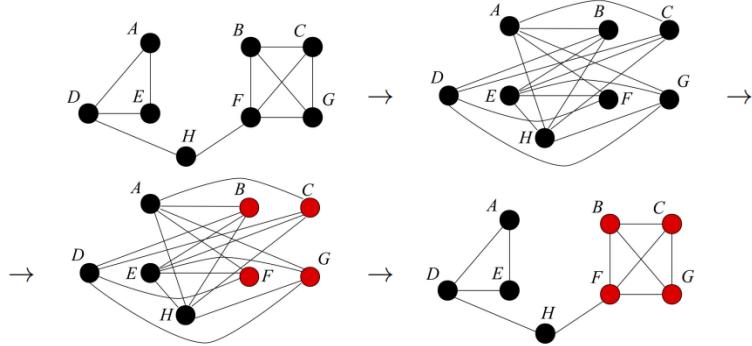


Figura 4.31: Una soluzione euristica MISP che da una soluzione euristica MCP

Questo processo può essere applicato anche nel verso opposto.

Anche i problemi SCP (*Set Covering Problem*) e VCP (*Vertex Covering Problem*) P hanno una relazione tra di loro, ma in maniera differente; ogni istanza del VCP può essere trasformata in un'istanza del SCP.

- Ogni arco i corrisponde ad una riga della matrice A .
- Ogni vertice j corrisponde ad una colonna A .
- Se l'arco i tocca il vertice j , l'insieme $a_{ij} = 1$, altrimenti $a_{ij} = 0$.
- Una soluzione ottimale del SCP da una soluzione ottimale del VCP.

	A	B	C	D	E	F	G	H
(A, D)	1	0	0	1	0	0	0	0
(A, E)	1	0	0	0	1	0	0	0
(B, C)	0	1	1	0	0	0	0	0
(B, F)	0	1	0	0	0	1	0	0
(B, G)	0	1	0	0	0	0	1	0
(C, F)	0	0	1	0	0	1	0	0
(C, G)	0	0	1	0	0	0	1	0
(D, E)	0	0	0	1	1	0	0	0
(D, H)	0	0	0	1	0	0	0	1
(F, G)	0	0	0	0	0	1	1	0
(F, H)	0	0	0	0	1	0	1	0

Figura 4.32: Una soluzione euristica SCP che da una soluzione euristica VCP

In questo caso non è semplice effettuare il procedimento inverso.

I problemi BPP (*Bin Packing Problem*) e PMSP (*Parallel Machine Scheduling Problem*) sono equivalenti, ma in una maniera più sofisticata:

- I task corrispondono a gli oggetti.
- Le macchine corrispondono ai container, ma ricordiamo che il BPP cerca di ottimizzare il numero di container (data una capacità), mentre il PMSP dato un numero di macchine cerca di ottimizzare il *tempo di completamento*.

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Partiamo da un istanza del BPP:

- Facciamo un assunzione sul numero di container ottimali, per esempio 3.
- Costruiamo una corrispondente istanza nel PMSP.
- Calcoliamo il tempo di completamento ottimale, per esempio 95; se eccede la capacità, incrementa l'assunzione fatta precedentemente (tipo 4 o 5). Nel caso contrario, decrementa l'assunzione fatta (2 o 1).

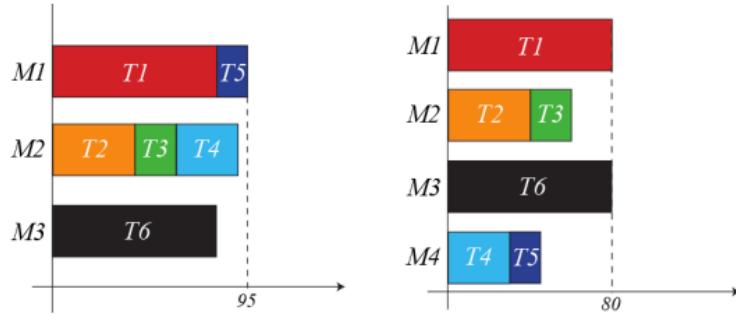


Figura 4.33: Una soluzione euristica PMSP che da una soluzione euristica BPP

Il processo inverso è possibile. I due problemi sono equivalenti, ma ognuno dei due deve venire risolto molteplici volte.

È importante sottolineare il fatto che in caso di **riducibilità**, una soluzione euristica all'interno di un istanza ridotta è una soluzione euristica per il problema originale; studiare le relazioni tra i problemi è importante anche senza pensare agli algoritmi.

4.8.5 Travelling Salesman Problem (TSP)

Dato:

- Un grafo **diretto** $G = (N, A)$
- Una funzione $c : A \rightarrow \mathbb{N}$ che provvede i costi per ogni arco.

Si vuole selezionare un ciclo di costo minimo che visiti tutti i nodi del grafo. Il **ground set** è l'insieme degli archi.

$$B \equiv A$$

La **regione di fattibilità** include i cicli che visitano tutti i nodi del grafo (**cicli Hamiltoniani**).

L'obiettivo è minimizzare il costo totale degli archi selezionati.

$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Come determinare quando un sottoinsieme è una soluzione fattibile? Quel sottoinsieme deve identificare un ciclo sul grafo ed ogni nodo deve avere esattamente un arco entrante ed uno uscente (ma sempre facente parte del sottoinsieme). Inoltre, una visita del grafo utilizzando gli archi del sottoinsieme dovrebbe visitare tutti i nodi, in altre parole non sono previste visite di G .

Cosa accade se si effettua una modifica di una soluzione fattibile? Questo dipende dal tipo della modifica effettuata sulla soluzione, potrebbe essere necessario ricalcolare la funzione obiettivo.

Risulta difficile trovare una soluzione fattibile? Trovare una soluzione che sia fattibile potrebbe essere altrettanto difficile; in generale un grafo Hamiltoniano (ovvero che presenta un ciclo Hamiltoniano) è un problema NP-completo, è di risoluzione banale solo nei grafi **completi**.

Consideriamo il seguente grafo diretto e pesato:

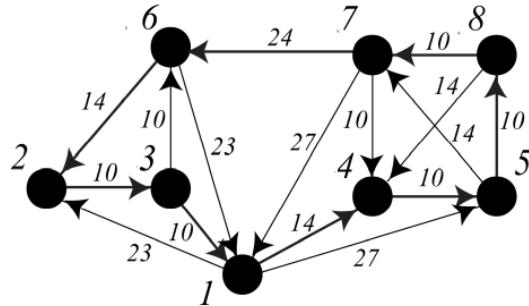


Figura 4.34: Dataset per il Travelling Salesman Problem

Consideriamo la prima soluzione proposta:

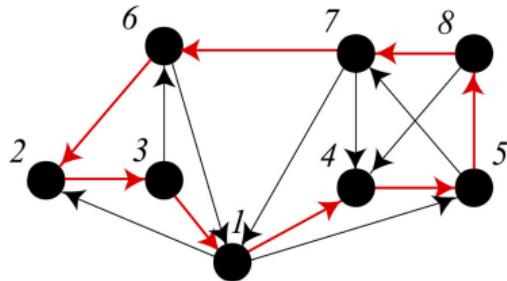


Figura 4.35: Prima soluzione del TSP

$$x' = \{(1,4), (4,5), (5,8), (8,7), (7,6), (6,2), (2,3), (3,1)\} \in X$$

$$f(x') = 102$$

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

La soluzione x' è una soluzione **fattibile**.

Consideriamo la seconda soluzione proposta:

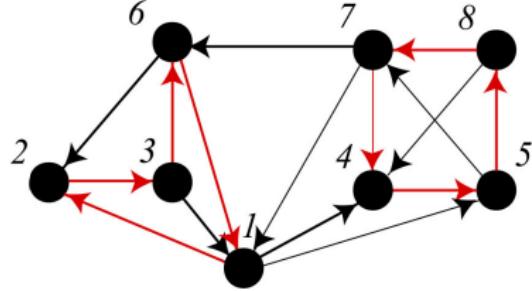


Figura 4.36: Seconda soluzione del TSP

$$x'' = \{(4,5), (5,8), (8,7), (7,4), (1,2), (2,3), (3,6), (6,1)\} \notin X$$

$$f(x'') = 106$$

La soluzione x'' è una soluzione **fattibile**.

4.8.6 Minimum Capacitated Spanning Tree Problem (MCSTP)

Dati:

- Un grafo indiretto $G = (V, E)$ con un vertice radice $r \in V$
- Una funzione $c : E \rightarrow \mathbb{N}$ che provvede il *costo* di ogni arco.
- Una funzione $w : V \rightarrow \mathbb{N}$ che provvede il peso di ogni vertice.
- Un numero $W \in \mathbb{N}$ che è la capacità di ogni sotto albero.

Si vuole selezionare un **minimo albero ricoprente** tale che ogni ramo (sotto albero rispetto alla radice) rispetti la capacità massima W . Il **ground set** è l'insieme degli archi.

$$B \equiv E$$

La **regione di fattibilità** include tutti gli alberi ricoprenti tali che il peso costituito dai vertici rispetti la capacità W . L'obiettivo è quello di minimizzare il costo totale degli archi selezionati.

$$\min_{x \in X} f(x) = \sum_{j \in x} c_j$$

Consideriamo il seguente grafo indiretto di pesi uniformi $w_i = 1 \forall i \in V$, e con una capacità massima dei sotto alberi $W = 3$.

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

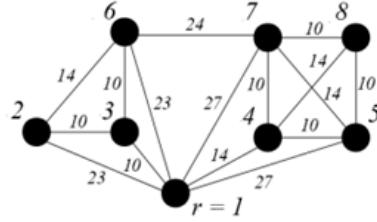


Figura 4.37: Dataset del Minimum Capacitated Spanning Tree Problem

La prima soluzione candidata:

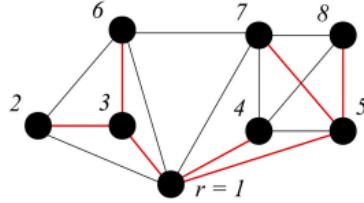


Figura 4.38: Prima soluzione del MCSTP

$$x' = \{(r,3), (3,2), (3,6), (r,4), (r,5), (5,7), (5,8)\} \in X$$

$$f(x') = 95$$

Questa è una soluzione **fattibile**, visto che ogni sotto albero non eccede la capacità W e i sottoinsiemi sono un minimo albero ricoprente del grafo.

La seconda soluzione candidata:

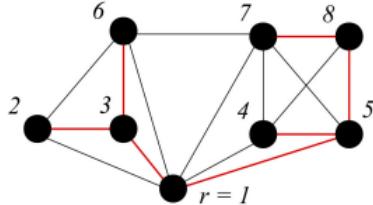


Figura 4.39: Seconda soluzione del MCSTP

$$x'' = \{(r,3), (3,2), (3,6), (r,4), (r,5), (5,7), (5,8)\} \notin X$$

$$f(x'') = 87$$

Questa **non** è una soluzione fattibile, visto che il sotto albero destro non rispetta la capacità massima (nonostante il sottoinsieme sia un albero ricoprente).

4 PROBLEMI DI OTTIMIZZAZIONE COMBINATORIA

Il costo delle operazioni principali cambia, il test di fattibilità richiede solamente la somma dei pesi, calcolare la funzione obiettivo richiede risolvere un problema MST (*Minimum Spanning Tree*).

La funzione obiettivo risulta:

- **Lenta da valutare**, calcolare un MST per ogni sottoinsieme.
- **Lenta da aggiornare**, ricalcolare un MST per ogni sottoinsieme modificato.

Se il grafo è **completo**, i test di ammissibilità sono:

- **Veloci da eseguire**, sommare i pesi dei vertici per ogni sotto albero.
- **Veloci da aggiornare**, sommare i pesi aggiunti e sottrarre quelli rimossi.

Visto che la funzione obiettivo è **additiva**, è abbastanza semplice da valutare, ma mentre è semplice da ricalcolare, è più difficile verificarne la fattibilità, eccetto per situazioni banali.

Trovare uno **albero ricoprente capacitato** è un problema fortemente NP-completo, quindi spesso è difficile trovare una soluzione fattibile a meno che il grafo sia completo. Dato un insieme di vertici, in ordine si controlla se la soluzione è una soluzione fattibile o meno, è necessario costruire la corretta rappresentazione dell'albero e poi visitare il sotto albero, sommare i pesi dei vertici, in una maniera simile ad una visita DFS.

Descrizione alternativa

4.8.7 Vehicle Routing Problem (VRP)

4.8.8 Interludio 6: combinare rappresentazioni alternative

Il CMSTP ed il VRP condividono una complicazione interessante: differenti definizioni del ground set B sono possibili e naturali.

- La descrizione come insieme di archi sembra preferibile per gestire la funzione obiettivo.
- La descrizione che utilizza un insieme di coppie del tipo $(vertice, albero) / (nodo/ciclo)$ sembra migliore per generare soluzioni ottime ed ha a che fare con la fattibilità.

Quale descrizione dovrebbe essere adottata? Quella che rende più semplice le operazioni più frequenti, oppure una possibilità risulta quella di utilizzare entrambe le rappresentazioni se le operazioni sono usate molto più frequentemente che quanto sono aggiornate, in maniera che il fardello di mantenerle aggiornate e consistenti sia accettabile.

Per riassumere, le domande che ci si deve chiedere a riguardo di un problema sono:

5 EFFICIENZA TEORICA

- Come si calcola la funzione obiettivo?
- Come provo che un sottoinsieme sia fattibile?
- Come trovo una soluzione fattibile?
- Come valutare il test di fattibilità?
- Cosa succede quando un cambiamento alla soluzione fattibile viene effettuato: è ancora fattibile? Sicuramente non lo è più? È necessario rivalutare da capo la funzione obiettivo o c'è una soluzione migliore?
- Quale è la definizione corretta di ground set? Sono qui presenti più definizioni possibili?
- Sono presenti relazioni tra questo ed un altro problema?

5 Efficienza teorica

La seconda parte di questo corso è dedicata alle caratteristiche degli algoritmi euristicci: abbiamo precedentemente descritto gli algoritmi euristicci come algoritmi che non provvedono sempre soluzioni corrette, ma che sono caratterizzati da due aspetti:

- Costa molto meno degli algoritmi corretti.
- *Spesso* restituisce qualcosa che è *vicino* alla soluzione corretta.

Considereremo questi due aspetti, **costi** e **qualità**, significa la distanza e la probabilità di ottenere una certa qualità e considereremo essi da due punti di vista:

- *Analisi a priori*, basata sulla teoria.
- *Analisi a posteriori*, basata sull'evidenza e sui dati empirici ottenuti dall'esecuzione dell'algoritmo su dataset appositi per test.

5.1 Problemi

Informalmente, un problema è una domanda su un sistema costituito da oggetti matematici. La stessa domanda può essere spesso posta su diversi sistemi simili.

- Un'istanza $I \in \mathcal{I}$ consiste in ogni specifico sistema riguardante la domanda.
- Una soluzione $S \in \mathcal{S}$ è una risposta corrispondente ad una delle istanze.

5 EFFICIENZA TEORICA

Per esempio: *n è un numero primo?*, questo è un problema con infinite istanze e due soluzioni.

$$\mathcal{I} = \mathbb{N}^+ \setminus \{1\} \text{ and } \mathcal{S} = \{\text{yes,no}\}$$

Formalmente, un problema è una funzione che relaziona le istanze e le soluzioni:

$$P : \mathcal{I} \rightarrow \mathcal{S}$$

Definire una funzione non significa sapere come calcolarla.

5.2 Algoritmi

Un algoritmo è una **procedura formale**, composta da passi elementari posti in una sequenza finita, ogni uno è determinato da un input e dai risultati dei passi precedenti.

Un algoritmo per un problema P è un algoritmo tale per cui un input $I \in \mathcal{I}$ restituisce una soluzione $S_I \in \mathcal{S}$.

$$A : \mathcal{I} \rightarrow \mathcal{S}$$

Un algoritmo definisce una funzione ed il modo per calcolarla, questo può essere:

- **Esatto**, se la funzione associata coincide con il problema.
- Altrimenti **euristico**.

Un algoritmo euristico è utile se risulta:

- **Efficiente**, significa che costa molto meno dell'algoritmo esatto.
- **Efficace**, significa che restituisce frequentemente la soluzione "vicina" a quella corretta.

5.3 Costi di un algoritmo euristico

Il costo di un algoritmo (entrambi i tipi) denotano il costo di computazione durante esecuzione:

- **Tempo**, richiesto per terminare la sequenza finita di passi elementari.
- **Spazio**, quello occupato in memoria dai risultati dei passi precedenti.

Il costo in tempo viene molto più discusso perché lo spazio è una risorsa rinnovabile, mentre il tempo non lo è. Utilizzare lo spazio richiede di utilizzare meno tempo possibile, inoltre è tecnicamente più semplice distribuire l'utilizzo di spazio che quello del tempo. Lo spazio ed il tempo sono parzialmente intercambiabili, è possibile ridurre il costo di uno incrementando l'utilizzo dell'altro.

5 EFFICIENZA TEORICA

5.3.1 Il tempo

Il tempo richiesto per risolvere un problema dipende da diversi aspetti:

- L'**istanza** specifica da risolvere.
- L'**algoritmo** utilizzato.
- La **macchina** che sta eseguendo l'algoritmo.

La nostra misura di tempo computazionale dovrebbe essere:

- **Sconnessa** dalla tecnologia, che sia la stessa su macchine differenti.
- **Coincisa**, che viene riassunta in una semplice espressione simbolica.
- **Ordinale**, che sia sufficiente per essere comparata con diversi algoritmi.

Il tempo computazionale in secondo per ogni istanza viola tutti i requisiti.

5.3.2 Complessità asintotica nel caso peggiore

La complessità asintotica di un algoritmo nel **caso peggiore** provvede una tale misura attraverso i seguenti passaggi:

1. Definire il tempo come un numero T di operazioni elementari eseguite.
2. Definire la dimensione di un istanza come opportuno valore di n .
3. Trovare il caso peggiore, ovvero il massimo valore di T su tutte le istanze di dimensione n .

$$T(n) = \max_{I \in \mathcal{I}_n} T(I), n \in \mathbb{N}$$

Ora la complessità in tempo è una funzione $T : \mathbb{N} \rightarrow \mathbb{N}$.

4. Approssimare $T(n)$ da sotto e/o da sopra con una funzione più semplice $f(n)$, questo considerando solamente il loro comportamento asintotico (per $n \rightarrow \infty$).
5. Collezionare le funzioni in **classi** con la stessa *funzione di approssimazione*.

5.3.3 Gli spazi funzionali Θ

$$T(n) \in \Theta(f(n))$$

Formalmente significa:

$$\exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} : c_1 f(n) \leq T(n) \leq c_2 f(n) \quad \forall n \geq n_0$$

Dove c_1, c_2 e n_0 sono indipendenti da n . $T(n)$ è "racchiusa" tra $c_1 f(n)$ e $c_2 f(n)$, per:

5 EFFICIENZA TEORICA

- Alcuni "piccoli" valori di c_1 .
- Alcuni "grandi" valori di c_2 .
- Alcuni "grandi" valori di n_0 .

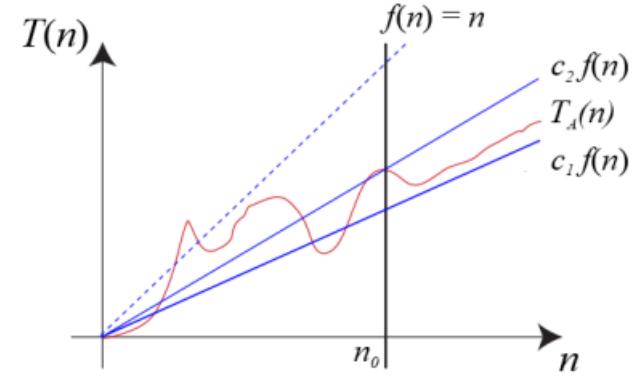


Figura 5.1: $f(n) = \Theta(T(n))$

Asintoticamente, la funzione $f(n)$ stima la funzione $T(n)$ per un fattore moltiplicativo: per grandi istanze, il tempo computazionale è al meno ed al più il valore della funzione $f(n)$.

5.3.4 Gli spazi funzionali O

$$T(n) \in O(f(n))$$

Formalmente significa:

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : T(n) \leq cf(n) \forall n \geq n_0$$

Dove c e n_0 sono indipendenti da n . $T(n)$ è "dominata" da $cf(n)$, per:

- Alcuni "grandi" valori di c .
- Alcuni "grandi" valori di n_0 .

Asintoticamente, la funzione $f(n)$ sovrasta la funzione $T(n)$ per un fattore moltiplicativo: per grandi istanze, il tempo computazionale è al più proporzionale al valore della funzione $f(n)$

5 EFFICIENZA TEORICA

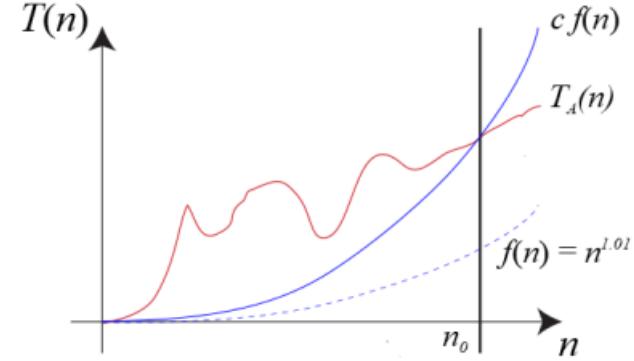


Figura 5.2: $f(n) = O(T(n))$

5.3.5 Gli spazi funzionali Ω

$$T(n) \in \Omega(f(n))$$

Formalmente significa:

$$\exists c > 0, n_0 \in \mathbb{N} : T(n) \geq cf(n) \forall n \geq n_0$$

Dove c e n_0 sono indipendenti da n . $T(n)$ "domina" $cf(n)$, per:

- Alcuni "grandi" valori di c .
- Alcuni "grandi" valori di n_0 .

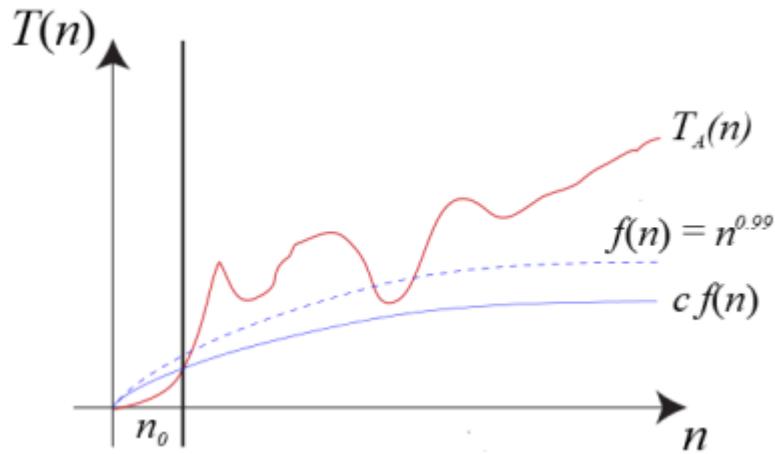


Figura 5.3: $f(n) = \Omega(T(n))$

5 EFFICIENZA TEORICA

Asintoticamente, $f(n)$ sovrasta $T(n)$ per un fattore moltiplicativo: per alcune grandi istanze, il tempo computazionale è almeno proporzionale al valore della funzione $f(n)$.

5.3.6 L'algoritmo esaustivo

Per i problemi di ottimizzazione combinatoria la dimensione di un'istanza può essere misurata dalla cardinalità del ground set.

$$n = |B|$$

L'algoritmo esaustivo:

- Considera ogni sottoinsieme $x \subseteq B$ tale che $x \in 2^{|B|}$.
- Testi la fattibilità (se $x \in X$) in un tempo $\alpha(n)$.
- In caso positivo, risolve la funzione obiettivo $f(x)$ in tempo $\beta(n)$.
- Se necessario, aggiorna il miglior valore trovato finora.

La complessità in tempo degli algoritmi esaustivi è :

$$T(n) \in \Theta(2^n (\alpha(n) + \beta(n)))$$

Questa risulta lo stesso esponenziale, anche se $\alpha(n)$ e $\beta(n)$ sono dei polinomi (caso più frequente). La maggior parte delle volte l'algoritmo esaustivo è **impraticabile**.

5.3.7 Complessità polinomiale ed esponenziale

Nei problemi di ottimizzazione combinatoria, la principale distinzione si trova tra:

- **Complessità polinomiale**, $T(n) \in O(n^d)$, dove $d > 0$ è costante.
- **Complessità esponenziale**, $T(n) \in \Omega(d^n)$, dove $d > 1$ è costante.

Nella prima famiglia fanno parte gli **algoritmi efficienti**, invece nella seconda quelli **inefficienti**. In generale, gli algoritmi euristicci sono algoritmi polinomiali per problemi dove l'algoritmo esatto risulta esponenziale.

5.3.8 Problemi di trasformazione e riduzione

Una relazione tra problemi permette la progettazione di algoritmi (*interludio 5*).

Progettare un algoritmo per **trasformazione**:

1. Data un'istanza di un problema P , detta I_P , trasformarla in una istanza di un altro problema Q , I_Q .
2. Data I_Q , applicare l'algoritmo A_Q per ottenere la soluzione S_Q .

5 EFFICIENZA TEORICA

3. Data S_Q , trasformiamola nella soluzione del problema P , S_P .

Progettare un algoritmo per **riduzione**:

1. Ripetere le trasformazioni 1, 2, 3 diverse volte correggendo I_Q basato sulle soluzioni $\{S_Q\}$ già ottenute.

I due algoritmi spesso hanno complessità simili: se A_Q è polinomiale (o esponenziale) e:

- Costruire I_Q impiegherà tempo polinomiale/esponenziale.
- Il numero di iterazioni sarà polinomiale/esponenziale.
- Costruire S_P impiegherà tempo polinomiale/esponenziale.

Quindi A_P è un algoritmo *polinomiale/esponenziale*.

Il punto è che se si ha una trasformazione nella quale la modifica dell'istanza è polinomiale, ed il tempo in cui la trasformazione è applicata è polinomiale, allora la complessità di tutto l'algoritmo dipenderà dalla complessità di A_Q .

Oltre alla complessità del caso peggiore La complessità nel caso peggiore ha diversi svantaggi:

- Cancella tutte le informazioni delle istanze più semplici, poiché considera solamente le istanza con grandi dimensioni.
- Fornisce una approssimativa sovrastima del tempo computazionale, che in alcuni rari casi è inutile. Per esempio, i problemi di *programmazione lineare*, nei quali i problemi hanno infinite soluzioni ma *finite* soluzioni base, ed ammettono un algoritmo che ha complessità esponenziale.

Tuttavia nella maggioranza dei casi gli algoritmi sono polinomi di bassa complessità.

Cosa se le istanze difficili sono rare nelle applicazioni pratiche? Per compensare, uno può investigare:

- La **complessità parametrizzata**, la quale introduce un nuovo rilevante parametro k (che è basato sulla dimensione di n), ed esprime il tempo come $T(n, k)$.
- La **complessità nel caso medio**, la quale si assume una probabilità di distribuzione su \mathcal{I} ed esprime il tempo come il valore atteso.

$$T(n) = E[T(I)|I \in \mathcal{I}_n]$$

5.4 Complessità parametrizzata

L'idea è che alcuni algoritmi non sono esponenziali in n ma sono esponenziali in un altro parametro k che è più piccolo di n .

5 EFFICIENZA TEORICA

Se entrambi k e n sono grandi, allora ovviamente l'algoritmo è inefficiente, ma se k è piccolo, allora l'algoritmo sarà efficiente e possibilmente anche polinomiale.

Perciò:

- Efficiente su istanze con k piccolo.
- Inefficiente su istanze con k grande.

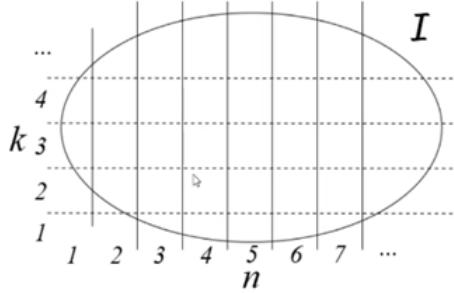


Figura 5.4: Insieme delle istanza infinitamente diviso dai parametri k ed n

Abbiamo tutti gli insiemi delle istanze del nostro problema, possiamo partizionare queste istanze con un numero infinito di fette per n e per l'altro parametro k . Possiamo considerare il nostro algoritmo.

Il parametro k può fare parte dell'input come la capacità, il massimo numero di letterali per formula (SAT), il numero di elementi non-zero (problema matrici numeriche), il massimo grado o il diametro (problemi con i grafi).

Stranamente il parametro aggiuntivo k può essere parte della soluzione (come la *cardinalità* nel VCP): in tale caso non puoi conoscere *a priori* se l'algoritmo è efficiente o meno, ma solo un'approssimazione può essere disponibile. Quindi se k è facente parte della soluzione, solo *a posteriori* sarà possibile sapere se si tratta di un algoritmo efficiente.

5.4.1 Bounded tree search (algoritmo con parametro)

Per esempio, il VCP

L'algoritmo esaustivo: per ogni uno dei 2^n sottoinsiemi di vertici, testare se coprono tutti gli archi, calcolare la cardinalità e mantenere il valore più piccolo.

$$T(n, m) \in \Theta(2^n(m + n)), \text{ dove } n = |V| \text{ e } m = |E|$$

Ma se noi già sappiamo una soluzione con $f(x) = |x| = k + 1$, possiamo cercare per una soluzione di k vertici, e progressivamente decrementare k (o ancora meglio utilizzare la **ricerca binaria** su k).

5 EFFICIENZA TEORICA

L'algoritmo naive: per ogni sotto insieme di k vertici, testare se copre tutti gli archi.

$$T(n, m, k) \in \Theta(n^k m)$$

Per una dato k fissato, l'algoritmo risulta di complessità polinomiale. Detto questo è possibile fare di meglio.

Bounded Tree Search per il VCP, un algoritmo migliore può essere basato sulla seguente proprietà:

$$x \cap (u, v) \neq \emptyset \quad \forall x \in X, (u, v) \in E$$

Ovvero, una qualsiasi soluzione fattibile include almeno un estremo (vertice) di ogni arco.

Allora l'algoritmo **Bounded tree search** (*ricerca delimitata da gli alberi*), vuole trovare x tale che $x \leq k$:

1. Scegliere un qualsiasi arco (u, v) , sia che $u \in x$ che $u \notin x$ e che $v \in x$ (ovvero che uno degli estremi non faccia per forza parte della soluzione).
2. Per ogni caso "aperto", rimuovere i vertici di x e gli archi che essi coprono.

$$V := V \setminus x, E := E \setminus \{e \in E : e \cap x \neq \emptyset\}$$

3. Se $|x| \leq k$ e $E = \emptyset$, x è la soluzione richiesta.
4. Se $|x| = k$ e $E \neq \emptyset$, non è presente alcuna soluzione.
5. Altrimenti ritorna al passo 1.

La complessità è $T(n, m, k) \in \Theta(2^k m)$, polinomiale in n ($m < n^2$). Per $n \gg 2$, questo algoritmo è molto più efficiente di quello *naive*.

Esempio del Bounded tree search

Nel seguente grafo con $n = 10, m = 16$, è presente una soluzione con $|x| \leq 3$? (quindi $k = 3$).

5 EFFICIENZA TEORICA

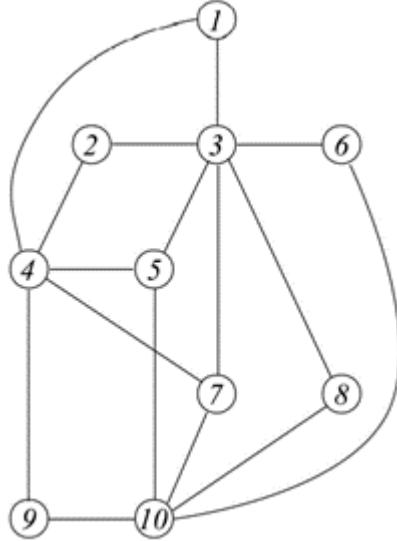


Figura 5.5: Stato iniziale del grafo (*Bounded Tree Search*)

Algoritmo esaustivo: $\Theta(2^n(m+n))$, con $2^n(m+n) = 1024 \cdot (16 + 10)$

Algoritmo naive: $\Theta(n^k m)$, con $n^k m = 1000 \cdot 16$

Bounded tree search algorithm: $\Theta(2^k m)$ con $2^k m = 8 \cdot 16$

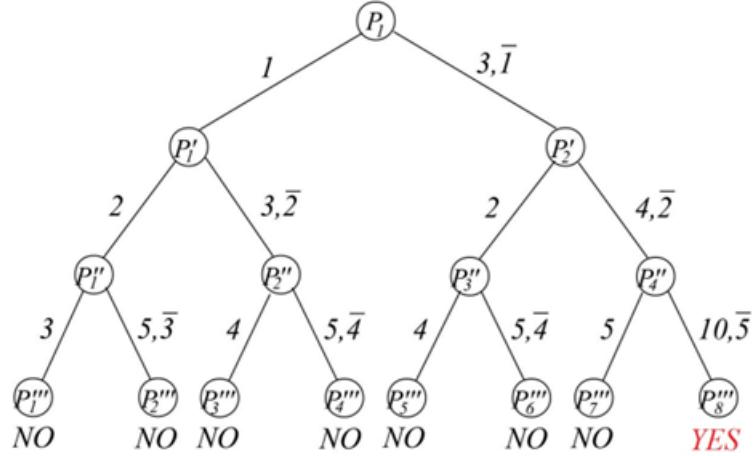


Figura 5.6: Soluzione del Bounded Tree Search (*ordinamento lessicografico*)

5 EFFICIENZA TEORICA

5.4.2 Kernelization - "Problem reduction" (algoritmo con parametro)

La *kernelizzazione* trasforma tutte le istanze di P in istanze più semplici di P , anziché istanze di un altro problema Q . Questa pratica è anche conosciuta con il nome di **problem reduction** (*riduzione del problema*).

Abbastanza spesso, in fatti, sono presenti delle proprietà molto utili che lo dimostrano:

- Esiste una soluzione ottimale che non include certi elementi di B (tali elementi possono essere rimossi).
- Esiste una soluzione ottimale che include certi elementi di B (tali elementi possono essere messi da parte ed aggiunti dopo).

In breve rimuove elementi di B senza modificare la soluzione. Possibili risultati utili sono:

- Un algoritmo esatto e polinomiale in n (complessità parametrizzata).
- Algoritmi esatti più veloci ed algoritmi euristicici.
- Migliori soluzioni euristiche.
- **Heuristic kernelization**, applica condizione rilassata sacrificando l'ottimalità.

Tornando al problema del VCP, vediamo la **Kernelization del VCP**:

Se $\delta_v \geq k + 1$, il vertice v appartiene ad una qualsiasi soluzione possibile di valore $\leq k$. Considerare che v ha $k + 1$ archi incidenti e che dovrebbe essere coperto da altrettanti vertici.

Algoritmo di kernelizzazione per mantenere solo i vertici della soluzione x con $|x| \leq k$:

- Iniziare allo step $t = 0$ con $k_0 = k$ ed un sottoinsieme di vertici vuoto $x_t := \emptyset$
- Impostare $t = t + 1$ ed aggiungere alla soluzione i vertici di grado $\geq k_t + 1$.

$$\delta_v \geq k_t + 1 \implies x_t := x_{t-1} \cup \{v\}$$

- Aggiornare k_t : $k_t := k_0 - |x_t|$
- Rimuovere i vertici con grado zero, quelli di x e degli archi coperti.

$$V := \{v \in V : \delta_v > 0\} \setminus x_t$$

$$E := \{e \in E : e \cap x_t = \emptyset\}$$

- Se $|E| > k_t^2$, non è presente alcuna soluzione fattibile (k_t vertici non sono abbastanza).

5 EFFICIENZA TEORICA

- Se $|E| \leq k_t^2 \implies |V| \leq 2k_t^2$, si applica l'algoritmo esaustivo.

La complessità è $T(n, k) \in \Theta(n + m + 2^{2k^2}k^2)$.

Per esempio, consideriamo il seguente grafo con $n = 10, m = 16$, è presente una soluzione con $|x| \leq k_0 = 5$?

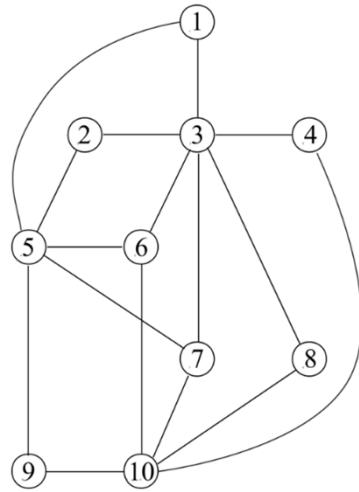


Figura 5.7: Grafo d'esempio per la Kernelization del VCP

Algoritmo esaustivo: $\Theta(2^n(m+n)) \implies T \approx 2^{10}(10+16) = 26624$

Algoritmo naive: $\Theta(n^k m) \implies T \approx 10^5 \cdot 16 = 16000000$

$\delta_3 \geq k_0 + 1 \implies x_1 := \{3\}$ rimozione degli archi incidenti ed avremo $k_1 = 4$.

5 EFFICIENZA TEORICA

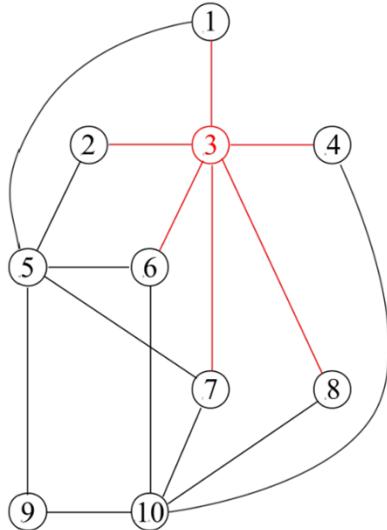


Figura 5.8: Rimozione degli archi incidenti con $k_1 = 4$

$\delta_{10} \geq k_2 + 1 \implies x_3 := \{5, 10\}$ rimozione degli archi incidenti ed avremo $k_3 = 2$.

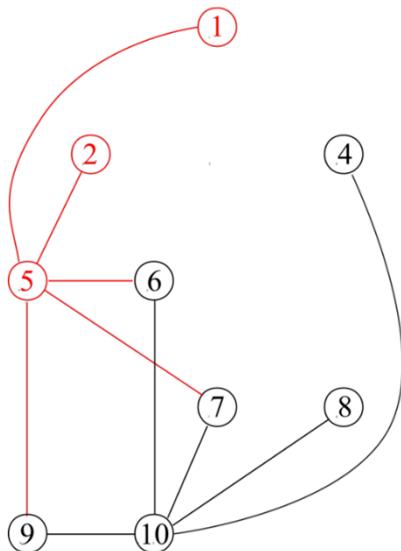


Figura 5.9: Rimozione degli archi incidenti con $k_2 = 3$

5 EFFICIENZA TEORICA

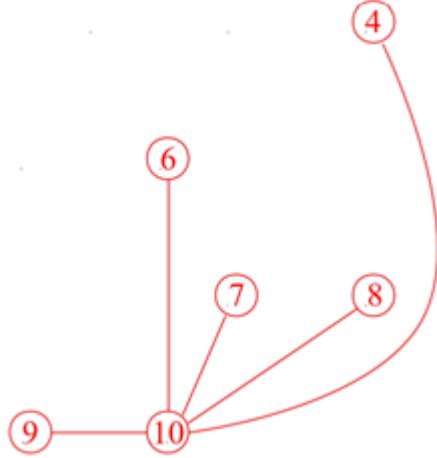


Figura 5.10: Rimozione degli archi incidenti con $k_3 = 2$

5.5 Complessità nel caso medio (average-case complexity)

Alcuni algoritmi sono inefficienti solo in un insieme di istanze ridotte, per esempio il *simplex algorithm* nel caso della programmazione lineare.

Gli studi teorici:

- Definiscono un modello probabilistico del problema. Il quale è una distribuzione probabilistica su $\mathcal{I}_n \forall n \in \mathbb{N}$
- Calcolano il valore stimato di $T(I)$

$$T(n) = E[T(I)|I \in \mathcal{I}_n]$$

Gli studi empirici:

- Costruiscono un *modello di simulazione* del problema, il quale è una distribuzione probabilistica di $\mathcal{I}_n \forall n \in \mathbb{N}$ teorico o empirico (costruito).
- Costruire un benchmark di istanze casuali relativo alla distribuzione.
- Applicare l'algoritmo e misurare il tempo richiesto.

5.6 Modelli probabilistici per matrici numeriche

Data una matrice binaria (m righe e n colonne):

- **Equiprobabilità**, elencare tutte le 2^{mn} matrici binarie e selezionare una delle matrici con probabilità uniforme.

5 EFFICIENZA TEORICA

- **Probabilità uniforme**, impostare ogni cella ad 1 con una data probabilità p

$$Pr[a_{ij} = 1] = p \quad (i = 1, \dots, m; j = 1, \dots, n)$$

Se $p = 0.5$, esso coincide con il modello di equiprobabilità, per un qualsiasi altro valore alcune istanze sono più simili di altre.

- **Densità fissa**, estrae δmn celle da mn con una probabilità uniforme e le imposta ad 1. Se $\delta = p$, esso assomiglierà al modello di probabilità uniforme, ma alcune istanze non potranno essere generate.

5.7 Modelli probabilistici per grafi

Dato un grafo casuale con un numero n di vertici:

- **Equiprobabilità**, elenca tutti i $2^{\frac{n(n-1)}{2}}$ grafi e seleziona quello con la probabilità uniforme.
- **Modello di Gilbert o probabilità uniforme $G(n, p)$**

$$Pr[(i, j) \in E] = p \quad (i \in V, j \in V \setminus \{i\})$$

Tutti i grafi con lo stesso numero di archi m hanno la stessa probabilità $p^m(1-p)^{\frac{n(n-1)}{2}-m}$ (diverso per ogni m). Se $p = 0.5$, esso coincide con il modello di equiprobabilità.

- **Modello di Erdos-Renyi $G(n, m)$** : estrae m coppie di vertici non ordinate su $\frac{n(n-1)}{2}$ con probabilità uniforme e crea un arco per ognuna. Se $p = \frac{2m}{n(n-1)}$, esso assomiglierà al modello con probabilità uniforme, ma alcune istanze non verranno generate.

5.8 Modelli probabilistici per funzioni logiche

Considerando una CNF casuale con un dato numero di variabili n :

- **Insieme con probabilità fissa**: elenca tutte le $\binom{n}{k}2^k$ formule con k letterali distinti e consistenti, ed aggiungo ognuno alla CNF con probabilità p .
- **Insieme con dimensione fissa**: costruisco m formule, aggiungo ad ognuna k letterali distinti e consistenti, estratti con probabilità uniforme. Se $p = \frac{m}{\binom{n}{k}2^k}$, esso assomiglierà al modello con probabilità fissa, ma alcune istanze non potranno essere generate.

5 EFFICIENZA TEORICA

5.9 Transizione di fase

Differenti valori (deterministici o probabilistici) dei parametri corrispondono a diverse regioni dell'insieme delle istanze.

Per i grafi:

- $m = 0$ e $p = 0$ corrisponde al grafo vuoto.
- $m = \frac{n(n-1)}{2}$ e $p = 1$ corrisponde al grafo completo.
- Valori intermedi corrispondono al grafo di densità intermedia (deterministicamente per m , probabilisticamente per p).

Per molti problemi la **performance** dell'algoritmo è decisamente differente in regioni diverse, riguardo a:

- **Tempo computazionale** (per algoritmi esatti ed euristici).
- **Qualità della soluzione** (per algoritmi euristici).

Spesso, la *variazione* di performance avviene all'improvviso in piccole regioni dello spazio dei parametri, come la *transizione di fase* all'interno di un sistema fisico.

Questo è utile per predire il comportamento di un algoritmo data un'istanza.

5.10 Transizione di fare per 3-SAT e Max-3-SAT

Data una CNF di n variabili e con formule logiche contenenti 3 letterali.

- 3-SAT: *è qui presente un assegnamento di verità che soddisfi tutte le formule?*
- Max-3-SAT: *quale è il massimo numero di formule soddisfacibili?*

Con l'aumentare del rateo formule-variabili $\alpha = \frac{m}{n}$:

- Le istanze soddisfacibili decrementano da quasi tutte (tante variabili per poche formule) a quasi nessuna (poche variabili per tante formule).
- Il tempo di calcolo per il 3-SAT incrementa fortemente e poi subito decrese, invece nel Max-SAT incrementa ulteriormente dopo la transizione.

5 EFFICIENZA TEORICA

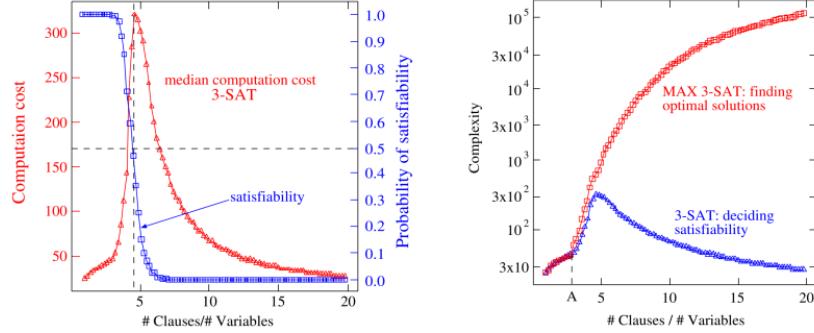


Figura 5.11: Plot delle fasi

Per $n \rightarrow \infty$ la transizione si concentra attorno $\alpha_c \approx 4.26$

5.11 La transizione di fase per il VCP

Il VCP esibisce una transizione di fase simile alla precedente quando $\frac{|x|}{|V|}$ incrementa.

- Il tempo computazionale prima esplode, dopo cade.
- Per $n \rightarrow \infty$ la transizione si concentra attorno al valore critico.

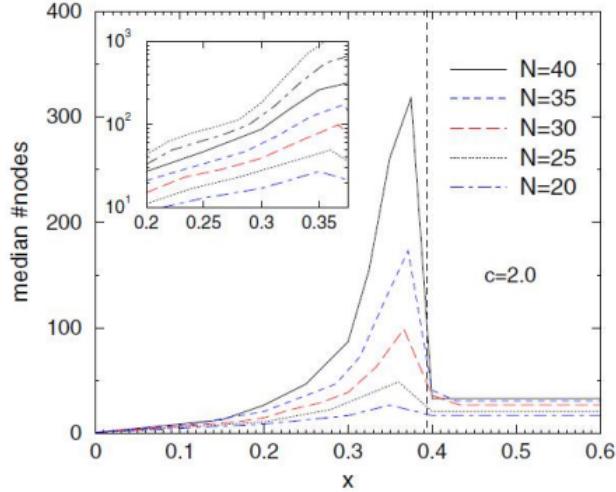


Figura 5.12: Fase del VCP

6 EFFICACIA TEORICA

Quando $\frac{|x|}{|V|}$ è piccolo, alcuni vertici sono chiaramente necessari, problema risolto. Quando $\frac{|x|}{|V|}$ è grande, molti vertici sono chiaramente necessari, problema risolto.

5.12 Costo computazionale degli algoritmi euristici

La complessità in tempo di un algoritmo euristico è solitamente:

- **Strettamente polinomiale** (con esponenti bassi).
- **Abbastanza robusta** rispetto ai parametri secondari.

Perciò, la stima nel caso peggiore è buona nella media dei casi. Le **metaeuristiche** utilizzano passi casuali o memoria:

- La complessità è ben definita per componenti singoli dell'algoritmo.
- La complessità globale non è chiaramente definita, in teoria potrebbe essere estesa *indefinitamente*, in pratica, è definita da una condizione imposta dall'utente.

Perché discutere ciò in un corso di algoritmi euristici ?

- Per guidare la ricerca dell'algoritmo corretto, un algoritmo corretto può essere efficiente in un caso specifico, ed inefficiente in un caso peggiore.
- Per mostrare che gli algoritmi euristici possono interagire proficuamente: poiché gli algoritmi euristici provvedono informazioni per migliorare gli algoritmi esatti (rendendoli più efficienti).
- Per mostrare che la kernelization migliora gli algoritmi euristici (diventano più efficienti ed efficaci).
- Per identificare *a priori* le istanze più ardue, certamente non tutti gli algoritmi hanno le stesse istanze "complesse".

6 Efficacia teorica

6.1 Efficacia di un algoritmo euristico

Un algoritmo euristico è utile se è:

- **Efficiente**, ovvero "*costa*" molto meno di un algoritmo esatto.
- **Efficace**, ovvero restituisce "*frequentemente*" una soluzione che è vicina a quella esatta.

6 EFFICACIA TEORICA

Adesso vogliamo discutere dell'efficacia degli algoritmi euristici, possiamo formalmente discutere questo concetto introducendo il concetto di distanza di una soluzione rispetto ad una ottimale e la frequenza (probabilistica) di ottenere una soluzione ottimale o quasi (rispetto ad una data distanza) rispetto ad una soluzione ottima. Queste caratteristiche possono essere combinate in una distribuzione a frequenza di soluzioni più o meno vicine alla soluzione ottimale (andrà introdotto un concetto di distanza).

L'efficacia di un algoritmo euristico può essere investigata in due modi:

- **Analisi teorica (*a priori*)**, provando che un algoritmo trova sempre o con una data frequenza soluzioni con una certa garanzia di qualità.
- **Analisi sperimentale (*a posteriori*)**, misurando le prestazioni dell'algoritmo su istanze di benchmark campionate per dimostrarne la presenza.

6.2 Distanza tra le soluzioni

Come detto precedentemente, dobbiamo dare un significato al concetto di distanza, e possiamo fornire molteplici interpretazioni differenti.

L'efficacia di un algoritmo di ottimizzazione euristico A è misurata dalla **differenza** tra il valore $f_A(I)$ (la funzione obiettivo sull'istanza I calcolata da A) e il valore ottimale $f^*(I)$ (la soluzione migliore per l'istanza I).

6.2.1 Differenza assoluta

Una possibile definizione iniziale tra due valori:

$$\tilde{\delta}_A(I) = |f_A(I) - f^*(I)|$$

Seppure questa definizione è molto naturale ed ovvia, è raramente utilizzata in pratica poiché dipende dall'unità di misura della funzione obiettivo: basa pensare di provare a minimizzare il tempo complessivo nel caso del TSP e che lo si misuri in giorni. Si potrebbe provare ad utilizzare ore, minuti o secondi anziché ed il valore di $\tilde{\delta}_A(I)$ dipenderà fortemente da questa scelta. Questo sarà valido quando la funzione obiettivo è un numero puro.

6.2.2 Differenza relativa

La seconda definizione è basata sull'idea che calcolare la differenza relativa rispetto alla soluzione ottimale.

$$\delta_A(I) = \frac{|f_A(I) - f^*(I)|}{f^*(I)}$$

Questo è un approccio molto frequente nell'analisi sperimentale ed il suo vantaggio è che nel caso in cui la funzione obiettivo abbia un'unità di misura, allora il rapporto (*rateo*) ottenuto sarà un numero *puro*.

6 EFFICACIA TEORICA

6.2.3 Rapporto di approssimazione

Un terzo approccio, ampiamente utilizzato, è il rapporto di approssimazione:

$$\rho_A(I) = \max \left[\frac{f_A(I)}{f^*(I)}, \frac{f^*(I)}{f_A(I)} \right] \geq 1$$

Il quale include sia il problema di minimizzazione che di massimizzazione: nel caso in cui sia un problema di minimizzazione, probabilmente $f^*(I) < f_A(I)$ e viceversa per i problemi di massimizzazione. Chiaramente è collegato alla differenza relazionale, in fatti nel caso dei problemi di minimizzazione:

$$\delta_A(I) = \rho_A(I) - 1$$

La relazione esiste anche per i problemi di massimizzazione.

6.3 Analisi teorica: garanzia di approssimazione

Per garantire *a priori* le prestazioni dell'algoritmo, l'idea è ancora una volta quella di considerare il caso peggiore, proprio come per l'efficienza. In generale quando si applica un algoritmo euristico il risultato $f_A(I)$ potrebbe essere un pessimo (distanziale) risultato rispetto alla soluzione ottimale $f^*(I)$, ma se l'algoritmo è buono la differenza non sarà molto grande;

La differenza tra $f_A(I)$ e $f^*(I)$ è generalmente illimitata, ma per alcuni algoritmi è limitata:

- **Approssimazione assoluta:**

$$\exists \tilde{\alpha}_A \in \mathbb{N} : \tilde{\delta}_A(I) \leq \tilde{\alpha} \quad \forall I \in \mathcal{I}$$

Significa che la distanza assoluta rispetto alla soluzione ottimale è limitata da un intero costante; un esempio è l'algoritmo di Vizing per la colorazione del grafo ($\tilde{\alpha} = 1$).

- **Approssimazione relativa:**

$$\exists \alpha_A \in \mathbb{R}^+ : \rho_A(I) \leq \alpha_A \quad \forall I \in \mathcal{I}$$

Significa che il rapporto di approssimazione $\rho_A(I)$ ha un limite superiore delimitato da una costante reale α_A . La cosa interessante è che la definizione può essere estesa a casi in cui una garanzia di approssimazione costante non può essere trovata considerando l'approssimazione come un valore ma come un'apposita funzione parametrizzata rispetto alla dimensione dell'istanza:

$$\rho_A(I) \leq \alpha_A(n) \quad \forall I \in \mathcal{I}_n, n \in \mathbb{N}$$

In conclusione, è importante sottolineare che mentre l'efficienza è necessariamente dipendente dalla dimensione dell'istanza, l'efficacia *potrebbe* non esserlo.

6 EFFICACIA TEORICA

6.3.1 Come ottenere una garanzia di approssimazione?

Fornite le basi teoriche, ora possiamo descrivere in maniera generale un metodo per introdurre e dimostrare un algoritmo con garanzia di approssimazione. Per i problemi di minimizzazione, si vuole dimostrare:

$$\exists \alpha_A \in \mathbb{R} : f_A(I) \leq \alpha_A f^*(I) \quad \forall I \in \mathcal{I}$$

Uno schema generalmente astratto è:

- Trovare un modo per costruire una sottostima $LB(I)$

$$LB(I) \leq f^*(I), I \in \mathcal{I}$$

- Trovare un modo per costruire una sovrastima $UB(I)$ che sia in relazione con il coefficiente α_A

$$UB(I) = \alpha_A LB(I), I \in \mathcal{I}$$

- Trovare un algoritmo A la cui soluzione non sia peggiore di $UB(I)$

$$f_A(I) \leq UB(I) \quad I \in \mathcal{I}$$

Se i tre passi sono stati completati e vengono trovati valori ed algoritmi adatti

$$f_A(I) \leq UB(I) = \alpha A LB(I) \leq \alpha A f^*(I) \quad \forall I \in \mathcal{I} \implies f_A(I) \leq \alpha_A f^*(I) \quad \forall I \in \mathcal{I}$$

La parte più complicata potrebbe essere il secondo passo, quindi affrontiamo mostrando un esempio.

Garanzia di approssimazione per il VCP: algoritmo 2-approssimato : dato un grafo indiretto $G = (V, E)$ trovare un sottoinsieme di vertici di cardinalità minima tale per cui ogni arco dell'arco del grafo sia incidente.

Definiamo:

- Un insieme “**matching**”, tale per cui sia l’insieme degli archi non adiacenti.
- Un insieme “**maximal matching**”, è un insieme di tipo *matching* tale che ogni altro arco del grafo è adiacente ad uno dei suoi archi.

Algoritmo di Matching:

1. Costruire un *maximal matching* $M \subseteq E$ scorrendo gli archi di E e includendo in M quelli che non sono adiacenti ad M (ora ogni arco di $E \setminus M$ è adiacente ad un arco di M).

6 EFFICACIA TEORICA

2. L'insieme di vertici *estremi* degli archi in *matching* è la soluzione del VCP

$$x_A := \bigcup_{(u,v) \in M} \{u, v\}$$

Può essere migliorata rimuovendo i vertici ridondanti.

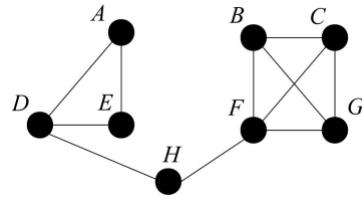


Figura 6.1: Algoritmo di Matching - Passo 0 (stato iniziale del grafo)

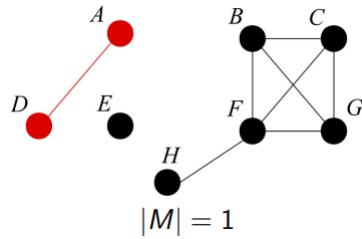


Figura 6.2: Algoritmo di Matching - Passo 1

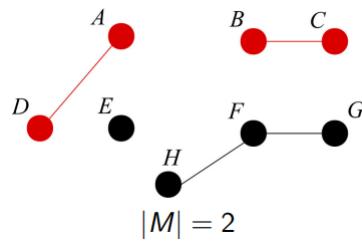
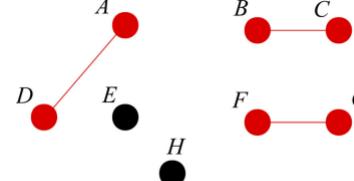


Figura 6.3: Algoritmo di Matching - Passo 2

6 EFFICACIA TEORICA



$$f_A = 2 \cdot |M| = 6 \text{ (and } G \text{ is redundant)}$$

Figura 6.4: Algoritmo di Matching - Passo 3

Torniamo adesso alla garanzia di approssimazione. Un possibile limite inferiore è quello di includere nella soluzione solamente uno dei vertici dell'arco nel *maximal matching*: considero solo gli archi nel *maximal matching* M .

È chiaro che il grafo consiste nei soli archi $e \in M$ ($G' = (V, M)$) solo metà dei vertici inclusi nella soluzione x_A copre gli archi in M . In altre parole, la copertura ottimale dei vertici degli archi in M è certamente più piccola della copertura ottimale di tutti gli archi in E .

Per fornire un esempio numerico, supponiamo che un certo grafo per il VCP abbia la soluzione ottimale $f^* = 5$ (quindi 5 vertici possono coprire tutti gli archi) e che l'algoritmo di matching trovi il matching massimale con 3 archi, quindi la soluzione dell'algoritmo sarà $f_A = 2 \cdot |M| = 6$. Però coprire tutti gli archi in M (considerare questo insieme come un grafo), basteranno $\frac{|M|}{2} = 3$ vertici!

Il valore $\frac{|M|}{2}$ è un possibile limite inferiore (o *lower bound*) per il nostro problema VCP:

$$LB(I) = \frac{|M|}{2} \leq f^*(I)$$

Ma c'è dell'altro: sappiamo che x_A viene calcolato prima che sia una soluzione attualmente fattibile ed è collegata a $LB(I)$:

$$f_A = 2 \cdot |M| = 2 \cdot LB(I) = UB(I)$$

Quindi abbiamo trovato sia il limite inferiore che quello superiore. Adesso forniamo una dimostrazione formale che l'algoritmo di matching è **2-approssimato**:

Dimostrazione algoritmo di Matching 2-approssimato, per prima cosa dimostriamo che la cardinalità dell'insieme "matching" sia una sottostima $LB(I)$: la cardinalità di una copertura ottimale per un qualsiasi sottoinsieme di archi $E' \subseteq E$ non supera quella di una copertura ottimale per E

$$|x_{E'}^*| \leq |x_E^*|$$

e la copertura ottimale di un qualsiasi insieme di matching M ha cardinalità $\frac{|M|}{2}$, quindi

$$|M| \leq |E| \implies |x_M^*| \leq |x_E^*| \implies \frac{|M|}{2} \leq |x_E^*| \forall M$$

6 EFFICACIA TEORICA

$$LB(I) = \frac{|M|}{2} \leq |x_e^*| \text{ è un limite inferiore valido}$$

In parole, visto che l'insieme M ha una cardinalità minore o uguale a quella di E , la soluzione ottimale (la quale ha cardinalità esatta di $\frac{|M|}{2}$) ha una cardinalità minore o uguale a quella della soluzione ottimale per l'insieme E , rendendola un valido limite inferiore.

Il fatto che $|M|$ stesso consiste in una sovrastima è provato dal fatto che la definizione dei vertici degli archi in M copre tutti gli archi in E , significa che l'insieme dei vertici M consiste in una soluzione fattibile (il che significa che è un limite superiore ed è un valore ottimale) e copre sia il "matching" che gli archi adiacenti. Perciò $|M| = UB(I) = 2 \cdot LB(I)$ è una sovrastima.

La prova si conclude con il fatto che l'algoritmo restituisce soluzioni di valore $f_A(I) \leq UB(I)$, visto che in M potrebbero essere presenti dei vertici *ridondanti* che possono essere rimossi. Questo implica che $f_A(I) \leq 2 \cdot f^*(I) \forall I \in \mathcal{I}$, che è $\alpha_A = 2$

Lo stretto legame tra stime inferiori e superiori, sono presenti delle istanze \bar{I} per il VCP tali che $f_A(\bar{I}) = 2 \cdot f^*(\bar{I})$ utilizzando l'algoritmo di Matching? Più in generale, sono presenti delle istanze \bar{I} tali che:

$$f_A(\bar{I}) = \alpha_A f^*(\bar{I})$$

e se così, come sono le istanze \bar{I} ? In altre parole, dopo aver dimostrato che $f_A(I)$ è in relazione con $f^*(I)$ attraverso α_A , questo è **costante**, **preciso** e **importante** o è "solo" un limite superiore? Lo studio delle istanze \bar{I} è utile per valutare se sono rare o frequenti e per introdurre modifiche *ad hoc* per migliorare l'algoritmo.

Garanzia di approssimazione per il TSP (disuguaglianza triangolare), consideriamo il TSP con delle assunzioni aggiuntive, ovvero che $G = (N, A)$ sia **completo** e che la funzione c sia simmetrica e soddisfi la **disuguaglianza triangolare**:

$$c_{ij} = c_{ji} \quad \forall i, j \in \mathbb{N}$$

e che

$$c_{ij} + c_{jk} \geq c_{ik} \quad \forall i, j, k \in \mathbb{N}$$

Il problema in generale, è fortemente NP-completo. Ma è anche fortemente NP-completo capire se le istanze hanno una soluzione fattibile (non quale sia fattibile), ovvero se è presente un ciclo Hamiltoniano che visiti tutti i nodi.

Sotto le precedenti assunzioni un algoritmo euristico può essere costruito con una garanzia di approssimazione associata ad esso.

Algoritmo Double-tree

1. Considera il grafo completo e indiretto G
2. Costruisci un minimo albero ricoprente $T^* = (N, X^*)$

6 EFFICACIA TEORICA

3. Fai una visita in pre-ordine di T^* e costruisci due liste di archi:

- (a) x sarà la lista degli archi utilizzata sia per la visita che per il backtracking: questo è un ciclo che visita ogni nodo, possibilmente molteplici volte.
- (b) x' sarà la lista degli archi che connettono i nodi in pre-ordine con il primo nodo: questo è un ciclo che visita ogni nodo esattamente una volta.

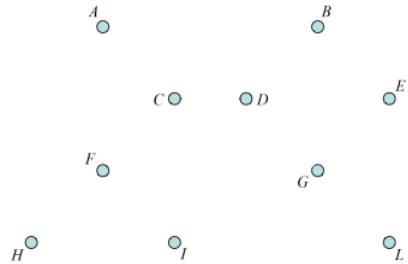


Figura 6.5: Double-tree per TSP - Passo 1 (grafo completo con archi omessi)

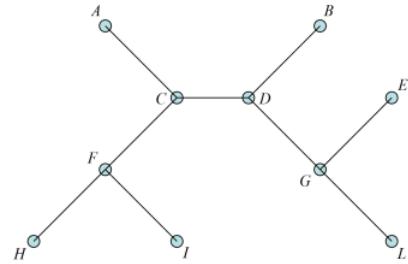


Figura 6.6: Double-tree per TSP - Passo 2 (Minimo Albero Ricoprente T^*)

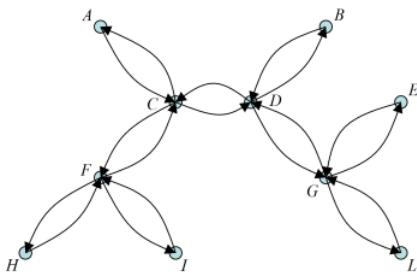


Figura 6.7: Double-tree per TSP - Passo 3.a (costruzione x)

6 EFFICACIA TEORICA

$$x = \{A, C, F, H, \textcolor{blue}{F}, I, \textcolor{blue}{F}, \textcolor{blue}{C}, D, G, L, \textcolor{blue}{G}, E, \textcolor{blue}{G}, \textcolor{blue}{D}, B, \textcolor{blue}{D}, \textcolor{blue}{C}, A\}$$

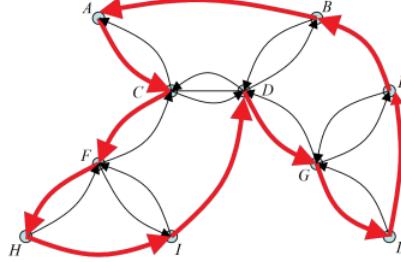


Figura 6.8: Double-tree per TSP - Passo 3.b (costruzione x')

$$x' = \{A, C, F, H, U, D, G, L, E, B, A\}$$

La struttura è la stessa del VCP: una sottostima è trovata assieme al fattore di moltiplicazione per costruire un limite superiore e quindi è provato che un algoritmo euristico può essere dominato da questo limite superiore, il che significa che l'algoritmo euristico è quasi sicuramente un fattore moltiplicato per la soluzione ottima. Andiamo a dimostrare che l'algoritmo double-tree è 2-approssimato.

Dimostrazione algoritmo Double-tree 2-approssimato, Rimuovendo un arco da un ciclo Hamiltoniano restituisce un percorso Hamiltoniano, il quale è necessariamente più economico. Visto che un cammino Hamiltoniano ricopre tutti i nodi, è effettivamente un albero ricoprente, e solitamente non di costo minimo. Quindi, un attuale minimo albero ricoprente è sicuramente un limite inferiore sul costo di un ciclo Hamiltoniano. La serie di nodi della visita DFS del MST (*Minimum Spanning Tree*) assieme con i suoi archi di back tracking genera un ciclo Hamiltoniano nel grafo originale, poiché è presente una coppia di archi diretti per ogni nodo (quindi ogni arco visitato nella DFS esiste nel grafo originale).

Il costo di questo ciclo Hamiltoniano è necessariamente una sovrastima, visto che il costo di tutti percorsi nel grafo rispetta la diseguaglianza triangolare; mentre la visita restituisce, per esempio il percorso $A \rightarrow B \rightarrow C$, il grafo rispetta l'uguaglianza triangolare scegliendo l'arco $A \rightarrow C$ la soluzione sarà di costo minimo. Quindi, visto che $LB(I)$ è uguale al costo del MST ed il ciclo ha esattamente il doppio degli archi del MST, abbiamo $UB(I) = 2 \cdot LB(I)$.

L'euristica è dominata da un limite superiore ed un qualsiasi circuito x ha:

$$f_A(I) \leq UB(I) \leq 2 \cdot LB(I) \leq 2 \cdot f^*(I) \quad \forall I \in \mathcal{I}$$

6 EFFICACIA TEORICA

6.4 Inapprossimabilità

Per un problema **inaprossimabile**, tutti gli algoritmi approssimati sono esatti, quindi alcuni problemi non possono essere approssimati a meno che alcune proprietà della complessità computazionale vengano verificata, cosa che è molto difficile che accada.

Per esempio, consideriamo la seguente famiglia di istanze del TSP che violano la disuguaglianza triangolare nella seguente maniera:

$$c_{ij} = \begin{cases} 0 & \forall (i, j) \in A_0 \subset A \\ 1 & \forall (i, j) \in A \setminus A_0 \end{cases}$$

ed il grafo è completo. Se una qualsiasi funzione di costo nullo ($f^*(I) = 0$) viene trovata, significa che la soluzione contiene solo archi in A_0 che compongono il ciclo Hamiltoniano. Quindi, in un senso, è presente un grafo non-completo $G(N, A_0)$ che ha un ciclo Hamiltoniano come soluzione fattibile, tale che è lo stesso per quello del grafo originale (questo significa che il grafo originale è il *completamento* del nuovo grafo). Questo è bello finché una soluzione per il grafo completo viene trovata con costo nullo, allora potrai veramente risolvere il problema del TSP nella forma decisionale anche sul grafo non completo, il quale è un problema fortemente NP-completo. Questo non può essere fatto esattamente, a meno che $P = NP$, il che è difficile. La funzione obiettivo di una qualsiasi istanza \bar{I} è:

$$\begin{cases} f^*(\bar{I}) = 0 & \text{Se } A_0 \text{ contiene un ciclo Hamiltoniano} \\ f^*(\bar{I}) \geq 1 & \text{Altrimenti} \end{cases}$$

Non possiamo trovare una garanzia di approssimazione sul valore α per un algoritmo polinomiale tale che:

$$f_A(I) \leq \alpha f^*(I) \quad \forall I \in \mathcal{I}$$

Poiché assumendo che sia possibile quello che accade è che se il grafo avesse un ciclo Hamiltoniano di costo zero, allora necessariamente l'algoritmo di approssimazione restituirà un costo:

$$f^*(\bar{I}) = 0 \Leftrightarrow f_A(\bar{I}) = 0$$

in parole povere, l'algoritmo di approssimazione risolve la versione decisionale del TSP nel caso di un'istanza con il grafo non-completo, questo è possibile risolvendo un problema fortemente NP-completo in tempo polinomiale e dimostrando che $P = NP$. Visto che questo è molto difficile che sia così, il problema il TSP è considerato un problema inaprossimabile.

6.5 Schemi di approssimazione

A volte è possibile trovare qualcosa meglio di una semplice approssimazione: per un qualsiasi problema difficile è presente un algoritmo esaustivo che

6 EFFICACIA TEORICA

fornisce la migliore approssimazione possibile garantendo $\alpha_A = 1$ (in quanto esatto), ciò richiede tempo esponenziale T_A .

Gli algoritmi approssimati solitamente una garanzia di approssimazione peggiore ($\alpha_A > 1$) ma essi potrebbero richiedere tempo polinomiale T_A .

Ora nel mezzo di questi due casi si possono trovare molte cose, significa che al posto di un singolo algoritmo polinomiale con tempo T_A e garanzia α_A potrebbe essere presente una completa famiglia di "differenti compromessi" tra efficienza ed efficacia, che incrementano la complessità computazionale e migliorano la garanzia di approssimazione.

- Migliori e migliori garanzie di approssimazione: $\alpha_{A_1} > \dots > \alpha_{A_r}$
- Peggiori e peggiori complessità computazionali: $T_{A_1} < \dots < T_{A_r}$

Tali famiglie di algoritmi sono chiamate **schemi di approssimazione**, i quali sono algoritmi parametrici A_α , che permettono di scegliere α . Un esempio di questo è il problema KP.

6.6 Oltre al caso peggiore

Proprio come per l'efficienza, l'efficacia ha dei metodi di misurazione che sono approcci differenti dal *caso peggiore*. Potresti avere un brutto caso peggiore senza approssimazione o un'approssimazione $\alpha = 1000000$ che restituisce una soluzione ottimale. Sono presenti differenti approcci.

6.6.1 Parametrizzazione

Anziché dividere le istanze per la dimensione, altri parametri k_i possono essere identificati in maniera che essi possano provvedere una garanzia di approssimazione che dipenda da k_i e non sia costante.

6.6.2 Caso medio

Assume una probabilità di distribuzione sulle istanze e ne calcola il fattore di approssimazione desiderato (l'algoritmo potrebbe avere delle brutte prestazioni solo su rare istanze).

6.6.3 Randomizzazione

Le operazioni dell'algoritmo dipendono non solo dall'istanza, ma anche dai numeri pseudocasuali, in maniera che la soluzione diventa una variabile casuale la quale possa essere investigata (la complessità in tempo potrebbe anch'essa essere casuale, ma solitamente non lo è).

Per un algoritmo casuale A , vengono considerate come variabili casuali $f_A(I)$ e $\rho_A(I)$. Un **algoritmo di approssimazione casuale** ha un rapporto di approssimazione il cui valore desiderato è limitato da una costante:

$$E[\rho_A(I)] \leq \alpha_A \quad \forall I \in \mathcal{I}$$

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

Considerando il problema Max-SAT: data una CNF, si vuole trovare l'assegnamento di verità per le variabili logiche tale che soddisfi un sottoinsieme di peso massimo di formule.

Allora un algoritmo puramente casuale assegna ad ogni variabile $x_j (j = 1, \dots, n)$:

- Un valore *False* con probabilità 0.5
- Un valore *True* con probabilità 0.5

Approssimazione casuale per il Max-SAT Sia $C_x \subseteq \{1, \dots, m\}$ il sottoinsieme di formule soddisfatte dalla soluzione x . Il valore oggettivo $f(x) = f_A(I)$ è il peso totale delle formule in C_x ed il valore desiderato è:

$$E[f_A(I)] = E \left[\sum_{i \in C_x} w_i \right] = \sum_{i \in C} (w_i \cdot Pr[i \in C_x])$$

Sia k_i il numero di letterali della formula $i \in C$ e $k_{min} = \min_{i \in C} k_i$

$$Pr[i \in C_x] = 1 - \left(\frac{1}{2} \right)^{k_i} \geq 1 - \left(\frac{1}{2} \right)^{k_{min}} \quad \forall i \in C$$

$$\implies E[f_A(I)] \geq \sum_{i \in C} w_i \cdot \left[1 - \left(\frac{1}{2} \right)^{k_{min}} \right] = \left[1 - \left(\frac{1}{2} \right)^{k_{min}} \right] \sum_{i \in C} w_i$$

e visto che $E[\rho_A(I)] = f^*(I)/E[f_A(I)]$ e $f^*(I) \leq \sum_{i \in C} w_i \quad \forall I \in \mathcal{I}$ uno ottiene:

$$E[\rho_A(I)] \leq 1 / \left[1 - \left(\frac{1}{2} \right)^{k_{min}} \right] \leq 2$$

7 Valutazione empirica delle prestazioni

Dopo la considerazione di un analisi algoritmica svolta *a priori*, la quale consiste nel dimostrare la presenza di alcune garanzie su alcuni aspetti dell'algoritmo che sono stati studiati come il costo computazione della qualità delle soluzioni, ora è possibile considerare l'analisi algoritmica *a posteriori*.

Questo perché l'analisi *a priori* è complicata su diversi aspetti: nell'analisi dell'efficienza il concetto è semplice mentre nell'analisi della qualità è molto più complicata, poiché i "passi elementari" degli algoritmi non hanno una relazione diretta con la qualità della soluzione. Inoltre, nel caso medio e della randomizzazione si richiede un trattamento statistico.

L'analisi teorica può esser una pratica **insoddisfacente** quando le conclusioni sono basate su **assunzioni non rappresentative**, come: un caso peggiore non frequente (molto difficile e su istanze molto rare), oppure una distribuzione probabilistica delle istanze irrealistica.

7.1 Analisi sperimentale

Il metodo sperimentale è l'approccio tipico che viene utilizzato nella scienza con l'eccezione della matematica, la quale è basata più su approcci formali, ma gli algoritmi sono l'eccezione all'eccezione, visto che solo alcune proprietà legate alle prestazioni degli algoritmi possono essere dimostrate e informazioni interessanti possono essere estrapolate dalle prestazioni pratiche.

Quindi abbiamo trovato un *isola empirica* in un mare di formalismo.

L'approccio empirico consiste nell'osservare la realtà ed effettuare alcune assunzioni su come la realtà "funzioni", questo formulando un **modello**. Dopo, una sequenza di passi viene ripetuta fino ad ottenere un **modello soddisfacente**:

1. Progettare esperimenti computazionali per validare il modello.
2. Eseguire gli esperimenti e collezionare i risultati.
3. Analizzare i risultati con metodi quantitativi.
4. Revisionare il modello in base ai risultati ottenuti.

7.1.1 Modello

Che cosa è un modello nel contesto dell'algoritmica? In fisica un modello è una legge che regola il comportamento di un fenomeno e, come analogia, nell'algoritmica un modello è una legge che ipoteticamente regola il comportamento dell'algoritmo stesso.

Le leggi possono essere del tipo: assumere che la complessità computazionale dell'algoritmo sia lineare, questo dipenderà da altri parametri come il massimo grado divertici del grafo, ecc...

Questo è qualcosa che viene assunto basato sulla conoscenza dell'algoritmo ma anche su un *trend* esposto dall'approccio empirico.

L'analisi sperimentale degli algoritmi mira:

1. Ottenere indici di efficienza ed efficacia compatti di un algoritmo.
2. Paragonare gli indici di algoritmi differenti per ordinarli.
3. Descrivere la relazione tra gli indici ed i valori parametrici delle istanze.
4. Suggerire miglioramenti negli algoritmi.

7.1.2 Benchmark

Visto che non tutte le istanze possono essere testate, viene definito un campione di riferimento (**benchmark sample**): un campione significativo deve rappresentare differenti

- Dimensioni, in particolare per l'analisi dei costi computazionali.

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

- Caratteristiche strutturale (*densità, grado, diametro, ...*).
- Tipi
 - di applicazione: logistici, telecomunicazioni, produzione, ...
 - di generazione: realistici, artificiali, trasformazioni di altri problemi,
...
 - di distribuzione probabilistica: uniforme, normale, esponenziale, ...

Cercare un benchmark sample che sia "equiprobabile" non è significativo perché:

- Le istanze di un insieme sono infinite.
- Gli insiemi infiniti non ammettono l'equiprobabilità (grande domanda della statistica).

Al contrario, noi possiamo:

- Definire classi finite di istanze che sono:
 - Sufficientemente difficili per essere istruttive.
 - Sufficientemente frequenti in applicazione di interesse.
 - Veloci abbastanza nella risoluzione, in modo da fornire dati sufficienti per le conclusioni.
- Estrarre benchmark samples da queste classi.

Riproducibilità

Il metodo scientifico richiede risultati riproducibili e controllabili. Quindi, se vengono fatte delle indagini, è necessario assicurarsi che altre persone siano in grado di replicare l'esperimento. Quindi, è necessario utilizzare **istanze pubblicamente disponibili** (o renderle pubblicamente disponibili), specificare i dettagli di implementazione, il linguaggio di programmazione e il compilatore dell'algoritmo implementato e i valori ambientali come la macchina utilizzata, il sistema operativo e la memoria disponibile.

7.2 Confronto di algoritmi euristici

Un algoritmo euristico è migliore di un altro altro quando simultaneamente

- Ottiene risultati migliori.
- Richiede meno tempo.

Algoritmi lenti con buoni risultati ed algoritmi veloci con brutti risultati non possono essere paragonati in una maniera significativa. In alcune situazioni specifiche il tempo computazionale può essere trascurato, precisamente quando non si vuole fare alcun confronto ma si cerca di fare solo una stima e anche quando si sa che il tempo di calcolo è più o meno lo stesso (per conosciuti motivi strutturali, ad es. due algoritmi uguali ma con qualche differenza nei parametri).

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

7.2.1 Descrivere le prestazioni di un algoritmo

Per descrivere le prestazioni di un algoritmo un **metodo statistico** può essere fornito. In particolare, l'idea è quella di modellare un algoritmo come se fosse un esperimento casuale in cui i risultati dipendono dalla scelta di un oggetto all'interno dello *spazio di campionamento* (sample space), definito come il campionamento delle istanze

$$\bar{\mathcal{I}} \subset \mathcal{I}$$

Il tempo computazionale $T_A(I)$ assieme alla differenza relativa $\delta_A(I)$ sono entrambe *variabili casuali*.

Le proprietà statistiche delle variabili casuali $T_A(I)$ e $\delta_A(I)$ descrivono le prestazioni dell'algoritmo A .

Quello che viene effettuato: anzichè considerare tutti i possibili valori del tempo computazionale e tutti i possibili valori della differenza relativa, si descrivono le variabili casuali con le loro proprietà statistiche, cosa che darà una descrizione statistica delle prestazioni dell'algoritmo.

7.3 Valutazione a posteriori dell'efficienza

7.3.1 Analisi del tempo computazionale (RTD diagram)

Il diagramma "Run Time Distribution" (RTD) è il grafico della funzione di distribuzione $T_A(I)$ su $\bar{\mathcal{I}}$.

$$F_{T_A}(t) = Pr[T_A(I) \leq t] \quad \forall t \in \mathbb{R}$$

Visto che $T_A(I)$ dipende fortemente dalla dimensione di $n(I)$, significa che diagrammi RTD significativi solitamente si riferiscono a benchmarks $\bar{\mathcal{I}}_n$ con un n fissato (e possibilmente fissati anche altri parametri suggeriti dall'analisi del caso peggiore).

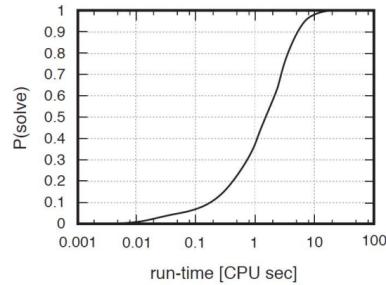


Figura 7.1: Diagramma Run Time Distribution

Se tutti i parametri influenti vengono identificati e fissati, il diagramma RTD degenera in una funzione a scala (ovvero, tutte le istanze richiedono lo stesso tempo).

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

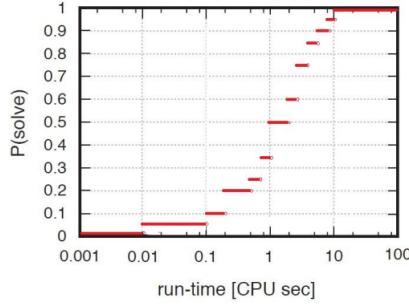


Figura 7.2: RTD degenerato in una *step function*

Il diagramma RTD è:

- **monotona non decrescente**, più istanze vengono risolte in più tempo.
- **graduale e continua verso destra**, il grafico aumenta ad ogni $T(I)$.
- **uguale a 0 per $t < 0$** , nessuna istanza è risolta in tempo negativo.
- **uguale ad 1 per $t \geq T(I)$** , tutti vengono risolti con il tempo più grande.
 $I \in \bar{\mathcal{I}}$

Per benchmark sample, il disegno sembra continuo, ma non lo è.

Costruzione del RTD:

- Eseguire l'algoritmo su ogni istanza $I \in \bar{\mathcal{I}}$
- Costruire l'insieme $T_A(\bar{\mathcal{I}}) = \{T_A(I) : I \in \bar{\mathcal{I}}\}$
- Ordinare $T_A(\bar{\mathcal{I}})$ non decrementando i valori $t_1 \leq \dots \leq t_{|\bar{\mathcal{I}}|}$
- Disegnare i punti $(t_j, \frac{j}{|\bar{\mathcal{I}}|})$ per $j = 1, \dots, |\bar{\mathcal{I}}|$ ed i segmenti orizzontali (chiusi a sinistra, aperti a destra).

7.3.2 Analisi del tempo computazionale (*scaling diagram*)

Lo **scaling diagram** descrive la dipendenza di $T(I)$ sulla dimensione di $n(I)$

- Genera una sequenza di valori di n ed un campion $\bar{\mathcal{I}}_n$ per ogni valore.
- Applica l'algoritmo ad ogni $I \in \bar{\mathcal{I}}_n \forall n$
- Disegna tutti i punti $(n(I), T(I))$ o i punti medi $\left(n, \frac{\sum_{I \in \bar{\mathcal{I}}_n} T(I)}{|\bar{\mathcal{I}}_n|}\right)$
- Assumere una **funzione interpolante**.
- Stimare i parametri numerici della funzione interpolante.

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

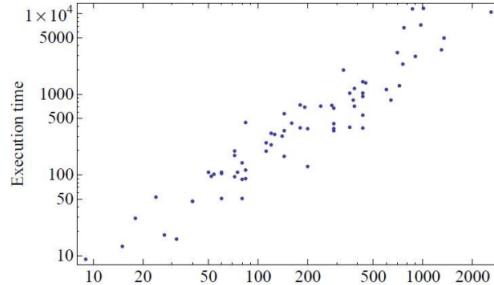


Figura 7.3: Stima dei parametri rispetto alla funzione interpolante

L’analisi fornisce empiricamente un caso medio di complessità, con fattori di molteplicità ben determinati (anziché c_1 e c_2), e non più grande del caso peggiore.

7.3.3 Interpolazione dello *scaling diagram*

La famiglia corretta di funzioni interpolanti può essere suggerita, da:

- Analisi teorica
- Manipolazione dei grafici

Solitamente l’**interpolazione lineare** è lo strumento corretto. Il dragamma di scaling si trasforma in una linea dritta quando

- Un algoritmo esponenziale è rappresentato da una **scala semi logaritmica** (il logaritmo viene applicato solo sull’asse del tempo).

$$\log_2 T(n) = \alpha n + \beta \Leftrightarrow T(n) = 2^\beta (2^\alpha)^n$$

- Un algoritmo polinomiale è rappresentato da una scala logaritmica (il logaritmo viene applicato su entrambi gli assi).

$$\log_2 T(n) = \alpha \log_2 n + \beta \Leftrightarrow T(n) = 2^\beta n^\alpha$$

7.4 Valutazione a posteriori dell’efficacia

7.4.1 Analisi della qualità della soluzione (*SQD diagram*)

Il diagramma **Solution Quality Distribution** (SQD) è il grafico della funzione di distribuzione $\delta_A(I)$ su $\bar{\mathcal{I}}$

$$F_{\delta_A}(\alpha) = \Pr[\delta_A(I) \leq \alpha] \quad \forall \alpha \in \mathbb{R}$$

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

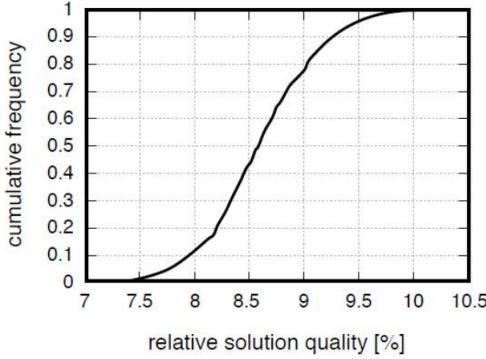


Figura 7.4: Diagramma *Solution Quality Distribution*

Per un qualsiasi algoritmo, la funzione di distribuzione di $\delta_A(I)$

- **Monotona non decrescente:** più casi vengono risolti con lacune peggiori.
- **graduale e continua a destra:** il grafo effettua un passo ad ogni $\delta(I)$
- **Uguale a 0 per $\alpha < 0$:** nessuna istanza viene risolta con lacune peggiori.
- **Uguale ad 1 per $\alpha \geq \max_{I \in \bar{\mathcal{I}}} \delta_I$:** tutte le istanze sono risolte all'interno della lacuna peggiore.

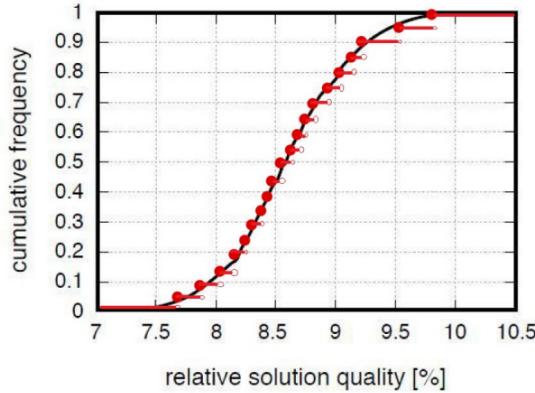
Se A è un:

- Algoritmo esatto, è una funzione a scala, uguale ad 1 $\forall \alpha \geq 0$
- Algoritmo $\bar{\alpha}$ -approssimato, è una funzione uguale ad 1 $\forall \alpha \geq \bar{\alpha}$

Costruzione del diagramma

- Eseguire l'algoritmo $\forall I \in \bar{\mathcal{I}}$
- Costruire l'insieme $\delta_A(\bar{\mathcal{I}}) = \{\delta_A(I) : I \in \bar{\mathcal{I}}\}$
- Ordinare $\delta_A(\bar{\mathcal{I}})$ per valori non decrescenti: $\delta_1, \dots, \delta_{|\bar{\mathcal{I}}|}$
- Disegnare i punti $(\delta_j, \frac{j}{|\bar{\mathcal{I}}|})$ per $j = 1, \dots, |\bar{\mathcal{I}}|$ ed i segmenti orizzontali (chiusi a sinistra, aperti a destra).

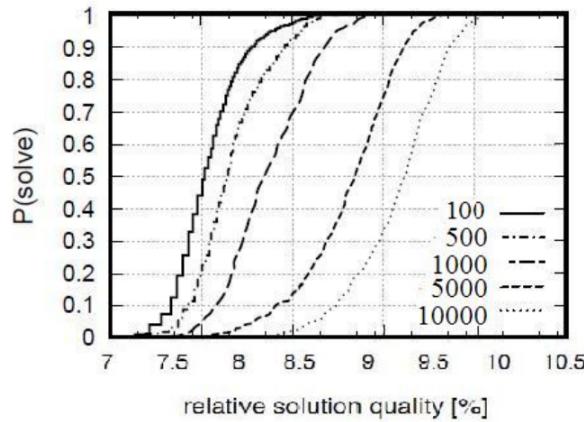
7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI



7.4.2 Diagrammi SQD parametrici

Dati i problemi teorici e pratici per costruire un campione significativo, il diagramma è parametrizzato rispetto a:

- Un parametro descrittivo delle istanze (dimensione, densità, ...)
- Un parametro di distribuzione probabilistica assunto per le istanze (varianza dei costi, ...)



Le conclusioni sono più limitate, ma il campione è più significativo, i **trend generali** possono essere evidenziati.

7.4.3 Confronto tra algoritmi utilizzando SQDs

Come determinare quando un algoritmo è migliore di un altro?

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

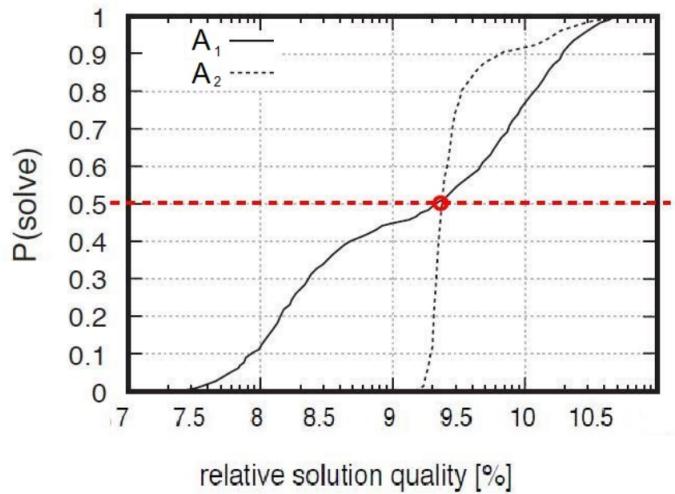
- **Dominanza stretta**, ottiene migliori risultati su tutte le istanze.

$$\delta_{A_2}(I) \leq \delta_{A_1}(I) \quad \forall I \in \mathcal{I}$$

- **Dominanza probabilistica**, la funzione di distribuzione ha valori più alti per tutti i valori di α .

$$F_{\delta_{A_2}}(\alpha) \geq F_{\delta_{A_1}}(\alpha) \quad \forall \alpha \in \mathbb{R}$$

Il seguente grafico non mostra alcuna dominanza, ma A_1 è meno "robusto" di A_2 , i risultati di A_1 sono più sparsi di A_2 .



7.5 Descrizione statistica compatta

Tutte le precedenti descrizioni prendono molto e sarebbe meglio avere dei numeri o una descrizione compatta delle prestazioni di un algoritmo: come un indice statistico riguardante la differenza relativa rispetto alla soluzione ottima. Questo coinvolge indici statistici classici di:

- posizione, come il **sample mean** (*mean* si riferisce alla media aritmetica di tutti i valori).

$$\bar{\delta}_A = \frac{\sum_{I \in \mathcal{I}} \delta_A(I)}{|\mathcal{I}|}$$

- dispersione, come il **sample variance**.

$$\bar{\sigma}_A^2 = \frac{\sum_{I \in \mathcal{I}} (\delta_A(I) - \bar{\delta}_A)^2}{|\mathcal{I}|}$$

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

Questi indici tendono ad essere molto influenzati dai **valori anormali** (*outliers*). Per esempio, un algoritmo che 9/10 fornisce la soluzione ottima ed 1/10 fornisce dieci volte la soluzione ottima, presenta chiaramente un valore anormale (questo non va bene). Utilizzando l'indice "sample mean" avverrà la somma delle differenze $\frac{0+0+\dots+9}{10} = 0.9$ restituendo un divario medio del 90%, il quale non è una buona rappresentazione.

Altri indici statistici sono più "*stabili*" e dettagliati:

- **sample median**
- **sample quantiles**

7.5.1 Boxplots

Sia il sample median che il quantiles possono essere rappresentati utilizzando una rappresentazione grafica chiamata **boxplot** (o diagramma a "*scatola e baffi*" o "*box and whiskers plot*").

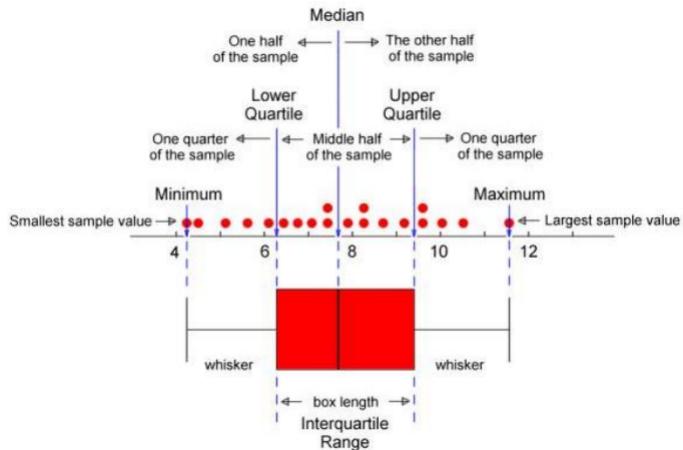


Figura 7.5: Esempio di boxplot

Supponiamo di avere 20 istanze nelle quali la relativa differenza spazia da $\approx 4\%$ a $\approx 11.5\%$, rispettivamente nel caso migliore e nel caso peggiore. All'interno del diagramma sono presenti cinque valori:

- **sample median** (*mediana*).
- **lower and upper sample quartiles** (*quartile minore e superiore*, rispetto alla mediana).
- **extreme sample values**, (*valori estremi*, escludendo gli *outliers*).

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

La **mediana** è il valore che si trova oltre metà delle istanze e sotto metà delle istanze: se il campione è costituito da un numero dispari di casi la mediana è il valore dell'elemento nel "mezzo", nel nostro caso la mediana è qualcosa tra la decima e l'undicesima istanza (un'altra opzione è quella di utilizzare proprio la decima istanza).

I quantili (considerando i *quartili*) tipici utilizzati in un boxplot sono il **primo ed il terzo**, cioè i valori che separano un quarto dell'osservazione dall'altra di tre quarti; Per esempio su 20 elementi il primo quartile corrisponde al quinto elemento mentre il terzo al quindicesimo.

Una volta trovati i primi tre valori sono necessari altri due valori che sono i campioni estremi, essi corrispondono al *minimo ed al massimo*.

Questo è un diagramma interessante che fornisce sia la posizione che la dispersione del benchmark, detto questo esattamente metà delle istanze cadrà nella "scatola" e le rimanenti tra la fine della scatola ed i baffi. Questo diagramma è una descrizione semplificata del SQD e, visto che due algoritmi possono essere paragonati utilizzando l'SQD, essi possono essere paragonati altrettanto utilizzando i relativi boxplot.

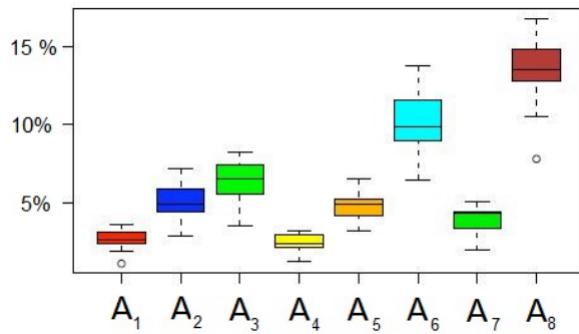


Figura 7.6: Confronto tra algoritmi attraverso i boxplot

Chiaramente, come mostrato in figura, il confronto è meno preciso di un diagramma SQD e meno informazioni possono essere estratte, tuttavia delle interessanti conclusioni possono essere fatte. In primis, prendendo sempre come esempio la figura sovrastante, gli 8 boxplot rappresentano la *differenza relativa* su un campione fisso di un benchmark, dove i cerchi sono gli outliers.

Se paragoniamo A_7 e A_8 , i due boxplot sono completamente separati, questo significa che tutte le istanze dell'algoritmo A_7 hanno valori migliori, questa è la definizione di **dominanza stretta**.

Invece, si parla di **dominanza probabilistica** quando ognuno dei 4 quartili di un boxplot si trova sotto l'altro boxplot con cui ci si sta confrontando, come per esempio tra A_2 e A_3 , ma questa **non è una condizione sufficiente** (solo necessaria). Sempre considerando lo stesso esempio potrebbe essere che un'istanza parta dal baffo inferiore di A_2 e tutte le altre si trovino esattamente nel

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

primo quartile mentre accade l'opposto per A_3 : in questo caso sicuramente non può essere considerata la dominanza probabilistica.

7.5.2 Relazione tra qualità e tempo computazionale

Un algoritmo euristico è migliore di un altro non quando restituisce risultati migliori ma quando viene eseguito in un tempo minore. Se questo non accade, non si può dire che uno domini l'altro.

Tuttavia, sono presenti molto algoritmi che non trovano una singola soluzione come gli algoritmi classici, ma quello che attuano è migliorare le soluzioni precedentemente trovate. Questo ne consegue in terminazioni premature.

In particolare gli algoritmi con *metaeuristiche* (passi casuali o meccanismi mnemonici) hanno un tempo computazione t fissato dall'utente e potenzialmente illimitato. Sia $\delta_A(t, I)$:

- La differenza relativa ottenuta da A al tempo t sull'istanza I .
- $+\infty$ se A non ha ancora trovato una soluzione fattibile nel tempo t .

Come funzione del tempo di calcolo t , $\delta_A(t, I)$ è:

- **monotona a scala non-crescente.**
- **costante dopo la terminazione regolare** $t \geq T(I)$ (se esiste).

Algoritmi randomizzati

Se gli algoritmi metaeuristici hanno passi casuali, allora la differenza relativa è funzione del seme casuale (*random seed*) $\omega \in \Omega$ rendendo $\delta_A(t, I, \omega)$. Quando si effettua il testing di un tale algoritmo, dovrebbe essere preso in considerazione che sono presenti due elementi casuali: l'istanza I estratta da $\bar{\mathcal{I}}$ ed il random seed.

Durante il test questi parametri possono essere combinati o distinti, significa che in un esperimento I può essere fissato and un gruppo di semi $\bar{\Omega}$ viene testato, o al contrario, ω è fisso ed un campione di istanze $\bar{\mathcal{I}}$ viene testato. I risultati su ω sono solitamente riassunti fornendo entrambe la differenza minima $\delta_A^*(t, I)$ ed il tempo totale $|\bar{\Omega}|t$, oppure la differenza media relativa $\bar{\delta}_A(t, I)$ ed il singolo tempo d'esecuzione t .

7.5.3 Classificazione

Ora abbiamo tutti gli elementi per fornire una **classificazione** abbastanza complicata per algoritmi euristicci, esatti ed approssimati. Questa vuole essere una classificazione di tutti i tipi di algoritmi per problemi di ottimizzazione combinatoria

La relazione tra la qualità della soluzione e tempo computazionale permette di classificare gli algoritmi in:

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

- **completi:** per ogni istanza $I \in \mathcal{I}$, si trova l'ottimo in un tempo finito (un nome diverso per gli *algoritmi esatti*).

$$\exists \bar{t}_I \in \mathbb{R}^+ : \delta_A(I, t) = 0 \forall t \geq \bar{t}_I, I \in \mathcal{I}$$

- **probabilisticamente ed approssimativamente completi** (*probabilistically approximately*): per ogni istanza $I \in \mathcal{I}$, si trova l'ottimo con una probabilità convergente ad 1, rispetto ad una differenza relativa uguale a 0, per $t \rightarrow \infty$ (molti algoritmi metaeuristici randomizzati). Questi algoritmi non forniscono una soluzione ottima in un tempo finito, ma lo faranno se verrà fornito *abbastanza tempo*.

$$\lim_{t \rightarrow +\infty} Pr[\delta_A(I, t) = 0] = 1 \forall I \in \mathcal{I}$$

- **essenzialmente incompleti:** per alcune istanze di $I \in \mathcal{I}$, si trova la soluzione ottima con una probabilità strettamente < 1 per $t \rightarrow +\infty$ (la maggior parte sono algoritmi greedy, algoritmi di ricerca locale, ...). Questi sono i "veri" algoritmi euristici.

$$\exists I \in \mathcal{I} : \lim_{t \rightarrow +\infty} Pr[\delta_A(I, t) = 0] < 1$$

Classificazione generalizzata

Un ovvia generalizzazione sostituisce la ricerca per l'ottimo con un dato livello di approssimazione:

$$\delta_A(I, t) = 0 \rightarrow \delta_A(I, t) \leq \alpha$$

- **algoritmi α -completi:** per ogni istanza $I \in \mathcal{I}$, trovano una soluzione α -approssimata in un tempo finito (questi essenzialmente è una descrizione formale per gli algoritmi α -approssimati).
- **algoritmi probabilisticamente ed approssimativamente α -completi:** per ogni istanza $I \in \mathcal{I}$, trovare una soluzione α -approssimata con una probabilità di convergenza ad 1 per $t \rightarrow +\infty$.
- **algoritmi essenzialmente α -completi:** per alcune istanze $I \in \mathcal{I}$ si trova una soluzione α -approssimata con una probabilità strettamente < 1 per $t \rightarrow +\infty$.

Concludendo, ogni algoritmo fornisce compromessi tra :

- una misura della qualità, descritta dalla soglia α .
- una misura del tempo, descritta dalla soglia t .

7.6 Diagrammi complessi

7.6.1 La probabilità di successo

Sia la **probabilità di successo** $\pi_{A,n} = \Pr[\delta_A(I, t) \leq \alpha | I \in \mathcal{I}_{n,\omega} \in \Omega]$ essere la probabilità che l'algoritmo A trovi in un tempo $\leq t$ una soluzione con una divergenza $\leq \alpha$ di istanze con dimensione n .

I due importanti parametri li abbiamo già visti sono la soglia t che è collegata all'efficienza dell'algoritmo, e la soglia α legata all'efficacia dell'algoritmo.

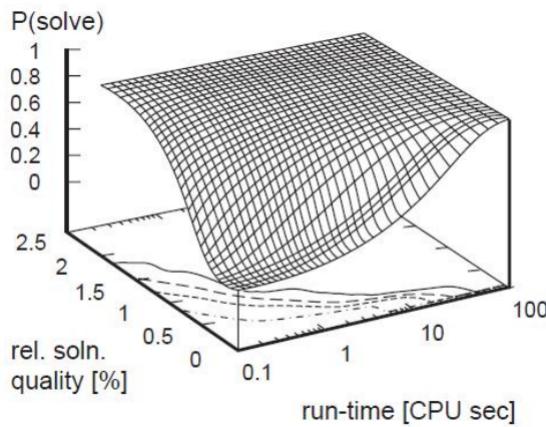


Figura 7.7: Esempio di diagramma per la probabilità di successo

Come rappresentazione stiamo considerando il diagramma tri-dimensionale poiché dipende da due variabili, la differenza relativa ed il tempo.

Da questo singolo diagramma è possibile estrarre altri tre classi di diagrammi "ausiliari", questo tagliando lungo uno dei tre assi, quindi rispetto al tempo, qualità e probabilità.

7.6.2 Qualified Run time Distribution diagram (QRTD)

I diagrammi QRTD descrivono il profilo del tempo richiesto per raggiungere uno specifico livello di qualità.

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

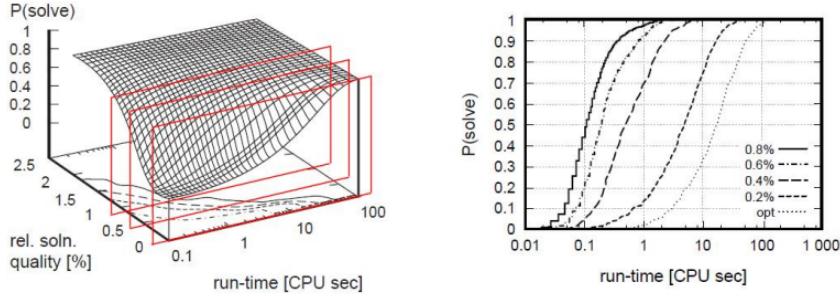


Figura 7.8: Un esempio di diagramma QRTD

Se si sta cercando di raggiungere una certa qualità, allora il plot del diagramma fornisce la probabilità di raggiungere tale qualità in un dato tempo. In base alla tipologia di algoritmo l'ottimalità verrà raggiunta in dato tempo o magari non raggiunta. Il QRTD utile quando il tempo computazionale non è una risorsa scarsa.

Se l'algoritmo è:

- completo, tutti i diagrammi raggiungono 1 in un tempo finito (ottimalità sicuramente raggiunta).
- $\bar{\alpha}$ -completo, tutti i diagrammi con $\alpha \geq \bar{\alpha}$ raggiungono 1 in tempo finito (l'ottimalità potrebbe essere raggiunta).
- $\bar{\alpha}$ -incompleti, tutti i diagrammi con $\alpha \leq \bar{\alpha}$ non raggiungono 1 (l'ottimalità certamente non è raggiungibile).

7.6.3 Timed Solution Quality Distribution diagram (TSQD)

I diagrammi TSQD descrivono il profilo del livello di qualità raggiunto in un dato tempo computazionale (qualità della soluzione relativa in un dato tempo).

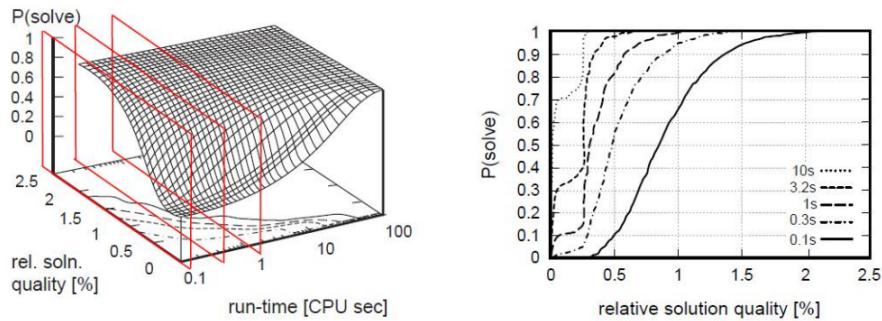


Figura 7.9: Un esempio di diagramma TSQD

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

Il TSQD utile quando il tempo computazionale non è una risorsa scarsa.
Se l'algoritmo è:

- completo, tutti i diagrammi con un sufficiente t sono funzioni a scala in $\alpha = 0$.
- $\bar{\alpha}$ -completo, tutti i diagrammi con un sufficiente t raggiungono 1 in $\alpha = \bar{\alpha}$.
- probabilisticamente approssimativamente $\bar{\alpha}$ -completo, i diagrammi convergono ad 1 per $\alpha = \bar{\alpha}$
- $\bar{\alpha}$ -completo, tutti i diagrammi mantengono una probabilità < 1 per $\alpha = \bar{\alpha}$

7.6.4 Solution Quality Statistics over Time diagram (SQST)

Finalmente possiamo disegnare le linee dei livelli associati con i differenti quantili, come rappresentato nella sottostante figura.

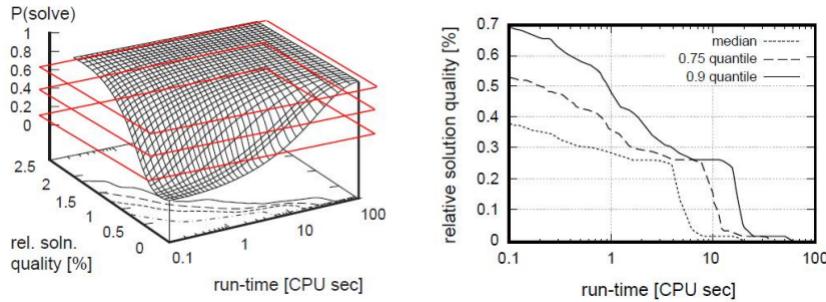


Figura 7.10: Un esempio di diagramma SQST

Essi descrivono il compromesso tra qualità e tempo computazionale, nel caso di un algoritmo *robusto* le linee dei livelli sono molto vicine tra di loro. Questo è un diagramma meno chiaro da leggere rispetto ad i precedenti. Supponiamo di essere interessati alla mediana (linea puntinata), significa che vogliamo i risultati che impiegano almeno metà del tempo. Aspettando ≈ 20 secondi una soluzione ottima potrà essere trovata (in metà del tempo rispetto ad un quantile, tipo il terzo), invece utilizzando un tempo minore una soluzione con qualità peggiore dovrà essere accettata.

7.7 Wilcoxon's test

I diagrammi ed i boxplot sono qualitativi, ma non sono veramente quantitativi. Per effettuare una valutazione quantitativa della differenza empirica tra i due algoritmi può essere utilizzato il **Wilcoxon's test** (il quale si concentra sull'efficacia seppur trascurando la robustezza).

1. $f_{A_1}(I) - f_{A_2}(I)$ è uan variabile casuale definita su uno spazio delle istanze \mathcal{I}

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

2. Formulare una **ipotesi nulla** (mancanza di relazione tra due fenomeni misurati) H_0 secondo la quale la mediana teorica di $f_{A_1}(I).f_{A_2}(I) = 0$
3. Estrarre un campione di istanze $\bar{\mathcal{I}}$ ed eseguire i due algoritmi su di esse, questo ci farà ottenere un campione di coppie di valori (f_{A_1}, f_{A_2}) .
4. Calcolare la probabilità p per ottenere i risultati osservati, o un risultato più "estremo", assumendo H_0 come vera.
5. Impostare un livello significativo \bar{p} , il quale è la **probabilità massima accettabile**
 - di rifiutare H_0 supponendo che sia vera.
 - di considerare due mediane identiche come differenti.
 - di considerare due algoritmi equivalenti come differentemente efficaci (in riferimento alla mediana della divergenza).

e rifiutare H_0 quando $p < \bar{p}$

I test statistici sono basati sulla formulazione di un'ipotesi della **ipotesi nulla** H_0 in maniera di comprendere la proprietà di osservare un certo comportamento empirico dato che tale ipotesi sia vera. In altre parole, la nostra ipotesi nulla è che metà delle volte f_{A_1} è meglio di f_{A_2} e metà delle volte è l'opposto. Questa è una buona descrizione di due algoritmi equivalenti.

Valori tipici di \bar{p} sono 5% e 1%. Se la probabilità calcolata è più piccola, allora H_0 è dimostrata falsa e rifiutata, e la mediana teorica tra gli algoritmi non sarà nulla.

7.7.1 Ipotesi sul Wilcoxon's test

Questo è un problema metodologico molto difficile. È un test **non parametrico**, il che significa che non si sta ipotizzando su nulla a riguardo della distribuzione dei valori testati. È utile valutare la prestazione degli algoritmi euristicci, perché la distribuzione dei risultati $f_A(I)$ è sconosciuta. È basato sulle seguenti ipotesi:

1. Tutti i dati sono misurati su (almeno) una scala ordinata. Il valore specifico non è importante, solamente la dimensione relativa.
2. I due data sets corrispondenti e derivano dalla stessa popolazione. Quindi applichiamo A_1 e A_2 alle stesse istanze estratte da \mathcal{I} .
3. ogni coppia di valori è estratta indipendentemente da gli altri. Le istanze sono generate indipendentemente l'una dall'altra.

La terza assunzione non è sempre verificata, quindi è necessario porre attenzione. Pensando a qualche problema complicato come il Vehicle Routing problem. Generare differenti famiglie di pesi sullo stesso grafo rende le istanze non proprio indipendenti l'una dall'altra.

Per eseguire il test devono essere seguiti i seguenti passi:

7 VALUTAZIONE EMPIRICA DELLE PRESTAZIONI

- Calcolare la differenza assoluta $|f_{A_1}(I_i) - f_{A_2}(I_i)| \forall I_i \in \bar{\mathcal{I}}$
- Ordinare quest'ultimi in ordine decrescente ed assegnare un **rango** (o *rank*) R_i a ciascuno di essi.
- Sommare separatamente i ranghi delle coppie con una differenza positiva e quelli con una differenza negativa. Se l'ipotesi nulla H_0 risulta vera, allora le due somme dovrebbero essere uguali.

$$\begin{cases} W^+ = \sum_{i:f_{A_1}(I_i) > f_{A_2}(I_i)} R_i \\ W^- = \sum_{i:f_{A_1}(I_i) < f_{A_2}(I_i)} R_i \end{cases}$$

- La differenza tra $W^+ - W^-$ permette di calcolare il valore di p : ognuna delle $|\bar{\mathcal{I}}|$ differenze può essere positiva o negativa, sono quindi presenti $2^{|\bar{\mathcal{I}}|}$ possibilità. p è la frazione con $|W^+ - W^-|$ uguale o maggiore rispetto al valore osservato.
- Se $p < \bar{p}$, la differenza è significativa e:
 - Se $W^+ < W^-$, l'algoritmo A_1 è migliore di A_2 .
 - Se $W^+ > W^-$, l'algoritmo A_1 è migliore di A_2 .

7.7.2 Calcolo di p

Il valore di p è solitamente:

- calcolato esplicitamente dall'enumerazione quando $|\bar{\mathcal{I}}| < 20$
- approssimato con una distribuzione normale quando $|\bar{\mathcal{I}}| \geq 20$

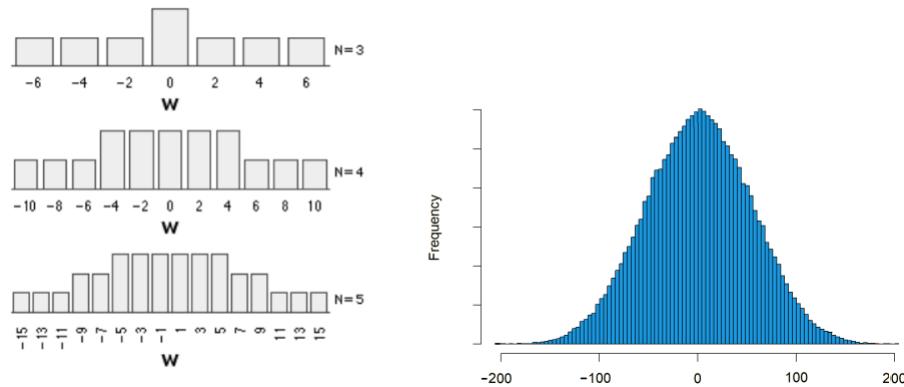


Figura 7.11: Distribuzione del Wilcoxon's test per svariate dimensioni di campioni

8 EURISTICHE COSTRUTTIVE

Per esempio, supponiamo di avere tre istanze I_1, I_2 e I_3 le quali restituiscono differenze con ranghi 1, 2 e 3. La situazione più sbilanciata è quando A_1 o A_2 è sempre migliore dell'altro, quindi quando ci sono tre differenze positive o tre negative; in altre parole potrebbe accadere che i ranghi 1, 2 e 3 siano tutti ranghi di differenze tutte positive o tutte negative.

Ogni valore quindi può essere "positivo" o "negativo" (nel senso che è un rango di una differenza positiva o negativa), quindi sono presenti $2^3 = 8$ possibilità.

Ciascuna delle quali è generalmente equiprobabile, quindi gli 8 casi possono essere numerati, come nella figura in alto a sinistra. In un caso tutti sono positivi e restituisce $W^+ - W^- = (1 + 2 + 3) - 0 = 6$; in un altro caso sono tutti negativi $W^+ - W^- = 0 - (1 + 2 + 3) = -6$, e così via; in due casi la differenza è 0 $W^+ - W^- = (1 + 2) - 3 = 3 - (1 + 2) = 0$.

Ci si aspetta dall'ipotesi nulla di essere in uno dei due casi dove la differenza è uguale a 0. Ma nel caso contrario, *quale è la probabilità?* Assumendo che le due popolazioni siano le stesse così che la mediana sia su 0, la probabilità di essere in una situazione di differenza **non equa o peggiore** corrisponde alla somma delle probabilità per ogni numerazione nel caso: per esempio, se la differenza restituita fosse 6 la sua probabilità è $\frac{1}{8}$. Se la differenza ottenuta è 4, la probabilità di ottenere ciò o peggio è $\frac{2}{8}$ e così via.

Quindi può essere calcolata attraverso l'enumerazione, o se sono presenti molte istanze, la legge dei grandi numeri dice che va bene approssimare con una distribuzione normale.

Possibili conclusioni

Il test di Wilcoxon può suggerire:

- che uno dei due algoritmi è significativamente migliore dell'altro.
- che i due algoritmi sono statisticamente equivalenti.

In entrambi i casi è importante "tenere un occhio" su p , poiché è una risposta stocastica.

Se il campione include istanze di tipi differenti, due algoritmi potrebbero essere complessivamente equivalenti, ma non equivalenti sulle singole istanze delle classi. Quindi se effettuo un test su $\delta_A(I)$ anziché $f_A(I)$? È una domanda aperta. I risultati possono essere differenti utilizzando $\delta_A(I)$ significa che dando un peso minore alle istanze che hanno una soluzione ottimale maggiore. Probabilmente, se il risultato del test è differente nei due casi, il testo è probabilmente non affidabile.

8 Euristiche costruttive

Adesso che abbiamo fornito un ampia panoramica dei problemi e dei metodi di valutazione degli algoritmi sia a priori che a posteriori, possiamo considerare la prima delle tre classi di euristiche, le **euristiche costruttive**.

8.1 Introduzione alle euristiche costruttive

Per comprendere questo tipo di euristica dobbiamo ricordarci che i problemi di ottimizzazione combinatoria ammettono delle soluzioni x che sono sempre sottoinsiemi di un dato ground set B , il quale è un insieme finito.

Il fatto che noi abbiamo sottoinsiemi suggerisce un possibile modo di costruire una soluzione, ed in particolare le euristiche costruttive consistono nell'iniziare da un insieme vuoto iniziale \emptyset e man mano aggiungere un elemento alla volta finché non si *intuisce* che non ha senso introdurre nuovi elementi, terminando così l'algoritmo. Analizziamo passo a passo questo processo, considerando il sottoinsieme $x^{(t)}$ aggiornato da una euristica costruttiva:

1. Si inizia da un insieme vuoto: $x^{(0)} = \emptyset$ (il quale è ovviamente sottoinsieme di una qualsiasi soluzione ottima). Sappiamo già che attraverso la kernelization o utilizzando procedure di riduzione può essere dimostrato che alcuni elementi del ground set sono necessariamente inclusi in almeno una soluzione ottima. In tali casi, potrebbe essere una buona idea iniziare ad includere nel sottoinsieme originale questi elementi "forzati".
2. Termina quando incontra una condizione di terminazione, il rationale di questa condizione consiste nel fatto che aggiungere un elemento nel sottoinsieme corrente non avrebbe alcun senso visto che non restituirebbe una soluzione ottima.
3. Seleziona il "*migliore*" elemento $i^{(t)} \in B$ tra quelli "*accettabili*" allo step t .
4. Aggiungi $i^{(t)}$ al sottoinsieme corrente $x^{(t)}$: $x^{(t+1)} := x^{(t)} \cup \{i^{(t)}\}$
5. Tornare al passo 2.

8.1.1 Il grafo di costruzione

Il grafo di costruzione (**construction graph**), è lo strumento di modellazione principale di questa costruttiva. Esso è un grafo diretto, costituito da nodi ed archi: i nodi di questo grafo costituiscono l'insieme \mathcal{F}_A (F sta per *find*), il quale è anche conosciuto come **spazio di ricerca** (o *search space*), il quale dipende dall'algoritmo A .

In altre parole \mathcal{F}_A dipende dal *problema* che si vuole risolvere, ma per lo stesso problema possono esserci differenti spazi di ricerca corrispondenti a differenti algoritmi, quindi differenti euristiche costruttive.

Lo spazio di ricerca è definito come la collezione di tutti i sottoinsiemi $x \subseteq B$ accettabili per l'algoritmo A .

$$\mathcal{F}_A \subseteq 2^B$$

Ogni *nodo* è un sottoinsieme ed alcuni saranno le soluzioni mentre altri no. Siamo interessati nei sottoinsiemi che sono accettabili ed utili. Gli *archi* connettono coppie di nodi, l'arco in se è la collezione di tutte le coppie $(x, x \cup \{i\})$ tali che $x \in \mathcal{F}_A, i \in B \setminus x$ e $x \cup \{i\} \in \mathcal{F}_A$. Quindi, gli archi connettono un

8 EURISTICHE COSTRUTTIVE

sottoinsieme della soluzione ad un sottoinsieme leggermente più grande che rappresenta l'estensione di un sottoinsieme ancora accettabile che include un nuovo elemento.

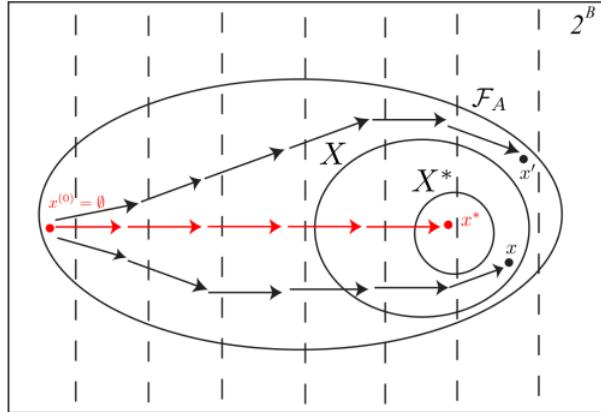


Figura 8.1: Il grafo di costruzione visita una sequenza di sotto insiemi che porta alla soluzione ottima

Questo grafo è un **DAG** (*Directed Acyclic Graph*). Ogni possibile esecuzione di A è un **percorso massimo** del grafo di costruzione (contando sia il sottoinsieme vuoto che un sottoinsieme la cui estensione non può essere accettata).

Il rettangolo del grafo di costruzione rappresenta le 2^B , ovvero tutti i possibili sottoinsiemi del ground set B . Questa collezione di sottoinsiemi è partizione in classi che dipendono dalla loro cardinalità, quindi la prima classe (quella più a sinistra) contiene solamente un possibile sottoinsieme, ovvero quello contenente l'insieme vuoto \emptyset (contenente tutti i sottoinsiemi di 0 elementi). Nella seconda classe dei sottoinsiemi possibili si trovano gli elementi singoli, poi le coppie, ecc. Sono presenti $n + 1$ classi che vanno da cardinalità 0 fino a n . All'interno dell'insieme potenza (o dei sottoinsiemi possibili), è presente lo spazio di ricerca degli algoritmi \mathcal{F}_A , il quale contiene alcuni sottoinsiemi che sono considerati "utili" per l'algoritmo: in particolare contiene l'insieme vuoto (l'algoritmo parte da quello), e altri insiemi che ha senso tenere conto.

Soltanamente sottoinsiemi che sono al di fuori dello spazio di ricerca sono quelli che non hanno senso di essere fissati, visto che non sono fattibili o perché non sono particolarmente "interessanti" ed ogni altro sottoinsieme che non può essere raggiunto da \mathcal{F}_A non è rilevante. Come prima l'insieme delle soluzioni fattibili viene denotato con X , e l'insieme delle soluzioni ottime X^* .

Nell spazio di ricerca è presente un insieme di archi che connettono coppie di nodi, il secondo nodo della coppia corrisponde al primo sottoinsieme della coppia più un certo elemento.

Come è possibile vedere dalle figure partire dal sottoinsieme vuoto è una soluzione accettabile, le soluzioni non *accettabili* si trovano al di fuori dello spa-

8 EURISTICHE COSTRUTTIVE

zio di ricerca, e di fatti non sono presenti archi che vanno al di fuori di questo. Quindi, ogni esecuzione di un algoritmo è un percorso massimo all'interno del grafo di costruzione, che parte dal sottoinsieme vuoto e raggiunge una soluzione, la quale idealmente dovrebbe essere ottimale o almeno fattibile. Nel caso ottimale, il percorso termina quando incontra una soluzione ottima, questo è il caso degli algoritmi di Dijkstra, Prim, Kruskal.

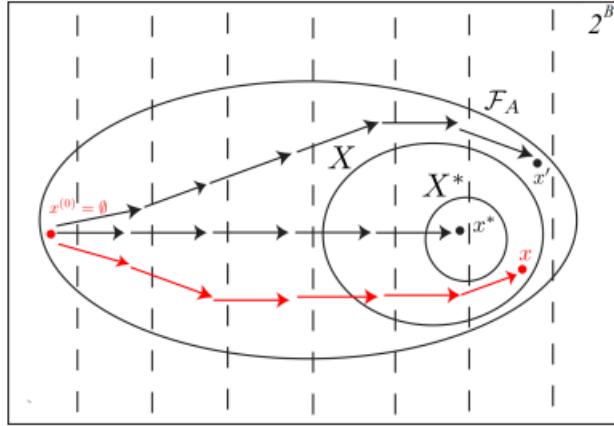


Figura 8.2: Il grafo di costruzione visita una sequenza di sotto insiemi che porta ad una soluzione fattibile ma non ottimale

Negli altri casi, come nella figura sovrastante, il percorso potrebbe non terminare in una soluzione ottima ma fattibile $x \in X$, questo è il caso degli algoritmi euristici come quelli usati per KP e MDP.

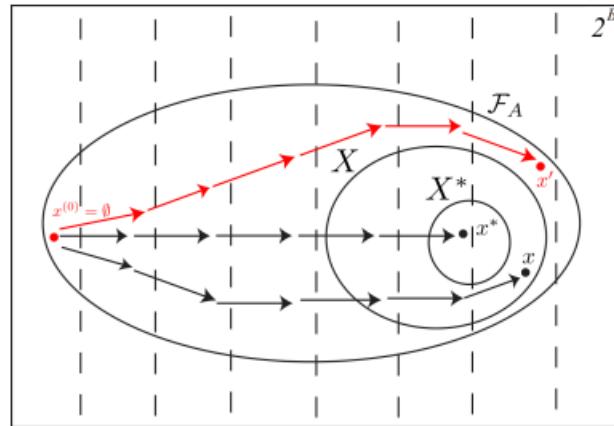


Figura 8.3: Il grafo di costruzione visita una sequenza di sotto insiemi che porta ad una soluzione non fattibile

8 EURISTICHE COSTRUTTIVE

Il percorso potrebbe terminare in una soluzione non fattibile $x \notin X$ (ma accettabile poiché presente nello spazio di ricerca), come per esempio il TSP su un grafo non completo.

Esempio di costruzione di grafi

Un esempio di grafo di costruzione è quello per il TSP per trovare il Nearest-Neighbour in un grafo completo. L'idea è quella di iniziare con un insieme di archi vuoto, quindi $x^{(0)} = \emptyset$, e sono presenti $n - 1$ archi uscenti.

La costruzione del grafo sarà un albero con $n - 1$ archi e n nodi; il numero di percorsi dall'insieme vuoto alle foglie sarà esattamente $(n - 1)!$, questo è il numero delle possibili esecuzioni dell'algoritmo.

Proprietà dello spazio di ricerca

- Include l'insieme vuoto $\emptyset \in \mathcal{F}_A$, visto che ogni algoritmo generalmente inizia da quello.
- In generale include l'insieme delle soluzioni fattibili $X \subseteq \mathcal{F}_A$, a volte alcune soluzioni possono essere rimosse da esso nel caso in cui risultino ovviamente non ottimali: per esempio l'SCP dove inizia con un insieme vuoto di colonne ogni colonna è a aggiunta, in un dato punto la soluzione fattibile viene raggiunta ma non ha senso aggiungere altre colonne perché i nuovi sottoinsiemi saranno fattibili ma anche più costosi.
- I sottoinsiemi di \mathcal{F}_A devono essere raggiungibili dal punto di partenza, altrimenti non avrebbe senso aggiungere una soluzione che non è raggiungibile dall'inizio. Un insieme candidato è la collezione delle **soluzioni parziali**, che sono le soluzioni generalmente non fattibili ma sono sottoinsiemi di soluzioni fattibili. Seguire una soluzione parziale fornisce almeno un percorso per una soluzione fattibile $\in X$.

Esempi di spazi di ricerca

Lo spazio di ricerca potrebbe includere tutte le soluzioni fattibili tali che:

$$\mathcal{F}_A \equiv X$$

come nel KP; potrebbe includere tutte le soluzioni parziali e tutte le soluzioni fattibili, tale che:

$$\mathcal{F}_A \equiv \bigcup_{x \in X} 2^x$$

per esempio nel MDP, nel quale non solo i sottoinsiemi dei k punti sono considerati ma anche quelli più piccoli o nell'algoritmo di Kruskal; lo spazio di ricerca potrebbe includere solo alcune soluzioni parziali:

$$\mathcal{F}_A \subset \bigcup_{x \in X} 2^x$$

8 EURISTICHE COSTRUTTIVE

un esempio di questo è l'algoritmo di PRim per MSP, il quale non considera tutte le soluzioni parziali ma considera un albero che è in aumento rimanendo connesso finché non diventa ricoprente. Le soluzioni possono essere definite come uno speciale sottoinsieme non sufficiente per garantire la fattibilità:

$$U_{x \in X} 2^x \mathcal{F}_A$$

un esempio di questo è il TSP, dove \mathcal{F}_A può essere definito su tutti i sottoinsiemi di archi che non includono diramazioni o sotto percorsi, non può essere definito come "tutti i sottoinsiemi che fanno parte delle soluzioni fattibili" visto che è un problema di decisione NP-completo.

8.2 Condizione di terminazione

Un euristica costruttiva A termina in due casi. Quando il sottoinsieme corrente x^t non può essere esteso senza che lasci lo spazio di ricerca \mathcal{F}_A :

$$x^{(t)} \cup \{i\} \notin \mathcal{F}_A \quad \forall i \in B \setminus x^{(t)}$$

o in maniera pressoché equivalente, non ha un arco uscente:

$$\Delta_A^+(x^{(t)}) = \{i \in B \setminus x^{(t)} : x^{(t)} \cup \{i\} \in \mathcal{F}_A\}$$

Sono possibili differenti comportamenti:

- Talvolta tutti i sottoinsiemi visitati sono fattibili (es. KP).
- Di solito l'ultimo sottoinsieme è l'unica soluzione fattibile.
- $x^{(t)}$ potrebbe muoversi dentro e fuori X (o X^*), seppur improbabile.

8.3 Struttura di algoritmi euristici costruttivi

Algorithm 1 Pseudo codice - Euristica Costruttiva

```

 $x := \emptyset$ 
 $x^* := \emptyset$ 
if  $x \in X$  then
     $f^* := f(x);$ 
else
     $f^* := +\infty;$ 
end if
while  $\Delta_A^+(x) \neq \emptyset$  do
     $i := \arg \min_{i \in \Delta_A^+(x)} \varphi(i, x);$ 
     $x := x \cup \{i\};$ 
    if  $x \in X$  and  $f(x) < f^*$  then
         $x^* := x;$ 
         $f^* := f(x);$ 
    end if
end while
Return( $x^*, f^*$ );

```

Il percorso (la sequenza di sottoinsiemi) visitato dall'algoritmo è determinato da:

- l'insieme $\Delta_A^+(x) \subseteq B \setminus x$, il quale è derivato dal grafo di costruzione.
- il criterio di selezione $\Phi_A : B \times \mathcal{F}_A \implies \mathbb{R}$ utilizzato per selezionare l'elemento i da aggiungere al sottoinsieme corrente $x^{(t)}$ per generare $x^{(t+1)}$, che può essere visto come una funzione peso su gli archi $(x, x \cup i)$.

La soluzione restituita è quella migliore visitata durante l'esecuzione (solitamente è l'ultima).

8.4 Definizione del grafo di costruzione

Idealmente, lo spazio di ricerca \mathcal{F}_A dovrebbe includere:

- Il sotto insieme vuoto $\in \mathcal{F}_A$
- Tutte le soluzioni possibili $X \subseteq \mathcal{F}_A$ (questo escludendo le soluzioni non ottime).
- solo sottoinsiemi raggiungibili da \emptyset (i sottoinsiemi non raggiungibili sono inutili).

8 EURISTICHE COSTRUTTIVE

I candidati naturali sono le soluzioni parziali (sottoinsiemi delle soluzioni fattibili), ma questi richiedono un test di inclusione veloce per rispondere alla domanda: *il sottoinsieme $x^{(t)}$ è una soluzione parziale?* o meglio $\exists x \in X : x^{(t)} \subseteq x$? o almeno dare un fast update test: sia $x^{(t)}$, allora $x^{(t)} \cup \{i\} \in \mathcal{F}_A$? Sfortunatamente, questo problema generalizza la fattibilità del problema: *è qui presente qualsiasi soluzione fattibile?*. Questo potrebbe essere un problema NP-completo, in questo caso si ha necessità di rilassare lo spazio di ricerca.

Se la funzione obiettivo potesse essere estesa da X a \mathcal{F}_A , allora sembrerebbe naturale utilizzare la funzione obiettivo come **criterio di selezione**.

$$\varphi(i, x) = f(x \cup \{i\})$$

Algorithm 2 Pseudo codice - Euristica Costruttiva (funzione obiettivo)

```

 $x := \emptyset$ 
 $x^* := \emptyset$ 
if  $x \in X$  then
     $f^* := f(x);$ 
else
     $f^* := +\infty;$ 
end if
while  $\Delta_A^+(x) \neq \emptyset$  do
     $i := \arg \min_{i \in \Delta_A^+(x)} f(x \cup \{i\});$ 
     $x := x \cup \{i\};$ 
    if  $x \in X$  and  $f(x) < f^*$  then
         $x^* := x;$ 
         $f^* := f(x);$ 
    end if
end while
Return( $x^*, f^*$ );

```

8.5 Il problema dello zaino frazionario (FKP)

Selezionando da un insieme di oggetti di volume identico un sottoinsieme di valore massimo tale che possa essere contenuto all'interno dello zaino, il quale dispone di una capacità massima limitata.

In questo problema la capacità impone un vincolo sulla cardinalità delle possibili soluzioni, esse sono $|x| \leq \lfloor \frac{V}{v} \rfloor$.

$$\varphi(i, x) = f(x \cup \{i\})$$

8 EURISTICHE COSTRUTTIVE

Algorithm 3 Pseudo codice - GreedyFKP

```

 $x := \emptyset$ 
 $x^* := \emptyset$ 
if  $x \in X$  then
     $f^* := f(x);$ 
else
     $f^* := +\infty;$ 
end if
while  $|x| < \lfloor \frac{V}{v} \rfloor$  do
     $i := \arg \max_{i \in B \setminus x} \phi_i;$ 
     $x := x \cup \{i\};$ 
    if  $x \in X$  and  $f(x) < f^*$  then
         $x^* := x;$ 
         $f^* := f(x);$ 
    end if
end while
Return( $x^*, f^*$ );

```

Definiamo $\mathcal{F}_A = X$, dove il sottoinsieme x può essere esteso finché $|x| < \lfloor \frac{V}{v} \rfloor$. Un qualsiasi elemento di $B \setminus x$ estende x . La funzione obiettivo è additiva, e quindi:

$$f(x \cup \{i\}) = f(x) + \phi_i \implies \arg \max_{i \in B \setminus x} f(x \cup \{i\}) = \arg \max_{i \in B \setminus x} \phi_i$$

L'ultimo sottoinsieme visitato è la migliore soluzione trovata. Consideriamo il seguente esempio:

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
$v_i = 1 \forall i \in B$						
$V = 4$						

L'algoritmo eseguirà i seguenti step:

- $x := \emptyset;$
- Dato che $|x| = 0 < 4$ allora calcola $i := a$ ed aggiorna $x := \{a\}$;
- Dato che $|x| = 1 < 4$ allora calcola $i := d$ ed aggiorna $x := \{a, d\}$;
- Dato che $|x| = 2 < 4$ allora calcola $i := c$ ed aggiorna $x := \{a, c, d\}$;
- Dato che $|x| = 3 < 4$ allora calcola $i := e$ ed aggiorna $x := \{a, c, d, e\}$;
- Dato che $|x| \geq 4$, allora termina.

Questo algoritmo trova sempre la soluzione ottima.

8 EURISTICHE COSTRUTTIVE

8.6 Il problema dello zaino (KP)

Select from a set of objects of different volume a maximum value subset which could be contained in a knapsack of limited capacity.

Algorithm 4 Pseudo codice - Greedy KP

```

 $x := \emptyset$ 
 $x^* := \emptyset$ 
 $f^* = 0$ 
while  $\exists i \in B \setminus x : v_i \leq V - \sum_{j \in x} v_j$  do
     $i := \arg \max_{\exists i \in B \setminus x : v_i \leq V - \sum_{j \in x} v_j} \phi_i;$ 
     $x := x \cup \{i\};$ 
end while
Return( $x, f(x)$ );

```

Definire $\mathcal{F}_A = X$, solo alcuni elementi di $B \setminus x$ estendono la fattibilità di c

$$\Delta_A^+(x) = \{i \in B \setminus x : \sum_{j \in x} v_j + v_i \leq V\}$$

La funzione obiettivo è additiva e quindi

$$f(x \cup i) = f(x) + \phi_i \implies \arg \max_{i \in \Delta_A^+(x)} f(x \cup \{i\}) = \arg \max_{i \in \Delta_A^+(x)} \phi_i$$

L'ultimo sottoinsieme visitato è la migliore soluzione trovata.

B	a	b	c	d	e	f
ϕ	7	2	4	5	4	1
v	5	3	2	3	1	1

L'algoritmo esegue i seguenti step:

- $x := \emptyset$;
- Dato che $\Delta_A^+ \neq \emptyset$ selezionare $i := \{a\}$; ed aggiornare $x := \{a\}$;
- Dato che $\Delta_A^+ \neq \emptyset$ selezionare $i := \{d\}$; ed aggiornare $x := \{a, d\}$;
- Dato che $\Delta_A^+ = \emptyset$, terminare.

Questo algoritmo non trova una soluzione ottima

8.7 Maximum Diversity Problem

Selezionare da un insieme di punti un sottoinsieme di k tale che la somma delle distanze fra le coppie di questo insieme sia massima.

8 EURISTICHE COSTRUTTIVE

Algorithm 5 Pseudo Codice - GreedyMDP

```

 $x := \emptyset$ 
while  $|x| < k$  do
     $i := \arg \max_{i \in B \setminus x} \sum_{j \in x} d_{ij};$ 
     $x : x \cup \{i\};$ 
end while
Return( $x, f(x)$ );

```

Definire lo spazio di ricerca \mathcal{F}_A come l'insieme di tutte le soluzioni parziali. Il sottoinsieme x può essere esteso finché viene soddisfatta la condizione del problema $|x| < k$. Un qualsiasi elemento di B estende x in una maniera fattibile. La funzione obiettivo è quadratica:

$$f(x \cup \{i\}) = f(x) + 2 \sum_{j \in x} d_{ij} + d_{ii} \implies \arg \max_{i \in B \setminus x} f(x \cup \{i\}) = \arg \max_{i \in B \setminus x} \sum_{j \in x} d_{ij}$$

L'ultimo sotto insieme visitato è la soluzione migliore (è l'unica fattibile) trovata.

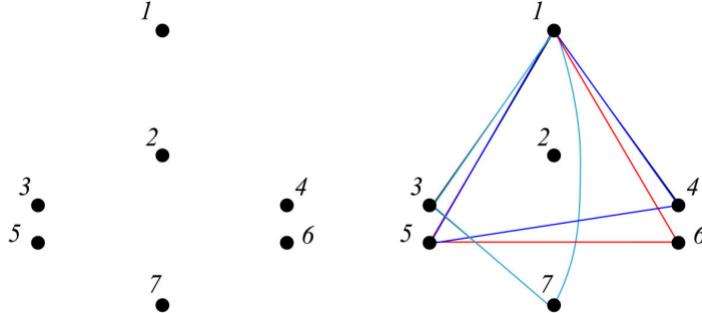


Figura 8.4: Maximum Diversity Problem

L'algoritmo ha due grossi svantaggi:

- Al primo passo, tutti i punti sono equivalenti.
- Il risultato finale non è ottimale anche se: il primo passo seleziona la coppia di punti più distanti o l'algoritmo viene ripetuto selezionando ogni punto come il primo.

8.7.1 Travelling Salesman Problem

Dato un grafo diretto ed una funzione di costo definita su gli archi, trovare il ciclo di costo minimo che sia in grado di visitare tutti i nodi del grafo.

Lo spazio di ricerca \mathcal{F}_A è definito come la collezione di tutti i sottoinsiemi di archi che formano un particolare sotto percorso e mantengono il grado dei

8 EURISTICHE COSTRUTTIVE

nodi ≤ 1 , per tutti i nodi (è un sovra insieme di tutte le soluzioni parziali). Il criterio di selezione è la funzione obiettivo, essendo additiva.

Algorithm 6 Pseudo Codice - GreedyTSP

```

 $x := \emptyset$ 
 $x^* := \emptyset$ 
 $f^* := \infty$ 
while  $\Delta_A^+(x) \neq \emptyset$  do
     $i := \arg \min_{i \in \Delta_A^+(x)}$ 
     $x : x \cup \{i\};$ 
end while
if  $x \in X$  then
     $x^* := x;$ 
     $f^* := f(x);$ 
end if
Return( $x^*, f^*$ );

```

Solo l'ultimo insieme visitato può essere fattibile (se trovato!). Per semplicità consideriamo un grafo simmetrico.

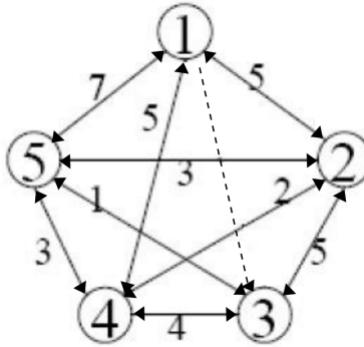


Figura 8.5: TSP grafo simmetrico

L'algoritmo eseguirà i seguenti passi:

- $x := \emptyset$;
- Visto che $\Delta_A^+(x) \neq \emptyset$ selezioniamo $i := (3, 5)$ ed aggiorniamo x ;
- Visto che $\Delta_A^+(x) \neq \emptyset$ selezioniamo $i := (2, 4)$ ed aggiorniamo $x((5, 3) \notin \Delta_A^+(x))$;
- Visto che $\Delta_A^+(x) \neq \emptyset$ selezioniamo $i := (5, 2)$ ed aggiorniamo $x((4, 2) \notin \Delta_A^+(x))$;

8 EURISTICHE COSTRUTTIVE

- Visto che $\Delta_A^+(x) \neq \emptyset$ selezioniamo $i := (4, 1)$ ed aggiorniamo x : notare che $(2, 5), (4, 5), (5, 4), (3, 4)$ e $(4, 3) \notin \Delta_A^+(x)$
- Visto che $\Delta_A^+(x) = \emptyset$, terminiamo.

L'algoritmo non trova una soluzione fattibile, ma una soluzione fattibile esiste: aggiungendo l'arco $(1, 3)$ (quello tratteggiato in figura) permette di trovare una soluzione fattibile anche nel caso in cui costi 100 (seppur non ottima).

8.8 Riassumendo

Un'euristica costruttiva A trova:

- Una soluzione ottimale quando $\Delta_A^+(x^{(t)})$ e $\varphi_A(i, x)$ garantiscono che il sottoinsieme corrente $x^{(t)}$ sia sempre incluso almeno in una soluzione ottima.
- Una soluzione fattibile quando $\Delta_A^+(x^{(t)})$ garantisce che il sottoinsieme corrente $x^{(t)}$ sia sempre incluso in almeno una soluzione fattibile.
- Un sottoinsieme generale quando queste proprietà sono perse ad un passo t .

Un algoritmo costruttivo ideale mantiene sempre una "via" aperta verso la soluzione ottima. In pratica, alcune di queste proprietà vengono perse durante qualche step dell'algoritmo (solitamente).

Caratteristiche rilevanti

Quali caratteristiche permettono un algoritmo costruttivo, di trovare l'ottimo?

- Uno spazio di ricerca uguale alla regione di fattibilità? No, perché questo viene tenuto
- Un problema vincolato dalla cardinalità? Questo spiegherebbe il fallimento del KP, ma non basta per il MDP e TSP.
- La presenza di una funzione obiettivo additiva?

Non esiste una caratterizzazione generale dei problemi risolti in maniera ottimale dagli algoritmi costruttivi. Sono però presenti caratterizzazioni per classi più grandi di problemi.

8.9 Una caratterizzazione nel caso additivo

Assumendo che:

- la funzione sia additiva:

$$\exists \phi : B \rightarrow \mathbb{N} : f(x) = \sum_{i \in x} \phi_i$$

8 EURISTICHE COSTRUTTIVE

- le soluzioni siano la **basi** (sottoinsiemi massimi) dello spazio di ricerca:

$$X = \mathcal{B}_{\mathcal{F}} = \{Y \in \mathcal{F} : \nexists Y' \in \mathcal{F} : Y \subset Y'\}$$

Questa è una situazione molto frequente come nel caso di KP, MAX-SAT, TSP ma non di MDP, SCP. In questo caso, l'algoritmo costruttivo trova sempre la soluzione ottima se e solo se (B, \mathcal{F}) è un *matroid embedding*. Visto che la definizione di tale insieme è molto complessa, concentriamoci sulla definizione di alcune strutture importanti (il concetto non verrà affrontato durante questo corso):

- **Greedoidi**
- **Matroidi**

8.10 Greedoidi

Un greedoide è una coppia di oggetti (B, \mathcal{F}) con $\mathcal{F} \subseteq 2^B$ è una coppia tale che soddisfi i seguenti assiomi:

- L'assioma **banale**: $\emptyset \in \mathcal{F}$
- L'assioma di **accessibilità**: se $x \in \mathcal{F}$ e $x \neq \emptyset$ allora $\exists i \in x : x \setminus \{i\} \in \mathcal{F}$. Un qualsiasi sottoinsieme accettabile può essere costruito aggiungendo elementi in un ordine adatto. Quindi deve essere possibile rimuovere 1 elemento tale che si rimanga ancora nello spazio di ricerca.
- L'assioma di **scambio**: se $x, y \in \mathcal{F}$ con $|x| = |y| + 1$. allora $\exists i \in x \setminus y$ tale che $y \cup \{i\} \in \mathcal{F}$. Ovvero. un qualsiasi sottoinsieme accettabile può essere esteso con un elemento adatto (per forza visto che rispettano entrambi il vincolo) di un qualsiasi altro sotto insieme di cardinalità più grande.

L'assioma di scambio implica che tutte le **basi** (ovvero i sottoinsiemi massimi) abbiano la stessa cardinalità. Tutte queste condizioni vengono mantenute nel caso FKP, e MST (sia Kruskal che Prim), mentre non vengono mantenute per il problema generale del KP e TSP. Vengono mantenute anche nel MDP, ma in quel caso la funzione obiettivo non è additiva. In generale gli algoritmi costruttivi non sono esatti sui greedoidi.

8.11 Matroidi

Un matroide è un insieme di sistema (B, \mathcal{F}) con $\mathcal{F} \subseteq 2^B$ tale che:

- L'assioma **banale**: $\emptyset \in \mathcal{F}$
- L'assioma di **eredità**: se $x \in \mathcal{F}$ e $y \subset x$ allora $y \in \mathcal{F}$. Ovvero che un qualsiasi sottoinsieme accettabile può essere costruito aggiungendo i suoi elementi in un qualsiasi ordine.

8 EURISTICHE COSTRUTTIVE

- L'assioma di **scambio**: se $x, y \in \mathcal{F}$ con $|x| = |y| + 1$. allora $\exists i \in x \setminus y$ tale che $y \cup \{i\} \in \mathcal{F}$. Ovvero. un qualsiasi sottoinsieme accettabile può essere esteso con un elemento adatto di un qualsiasi altro sotto insieme di cardinalità più grande.

L'assioma di eredità è una versione più forte dell'assioma di accessibilità dei greedoidi. Viene mantenuto nello spazio di ricerca dell'algoritmo di Kruskal per il problema del MST. Ma non viene mantenuto nello spazio di ricerca dell'algoritmo di Prim.

$$x = \{(A, D), (D, H), (E, F), (B, F), (C, G)\}$$

$$y = \{(A, E), (B, H), (E, F), (E, H)\}$$

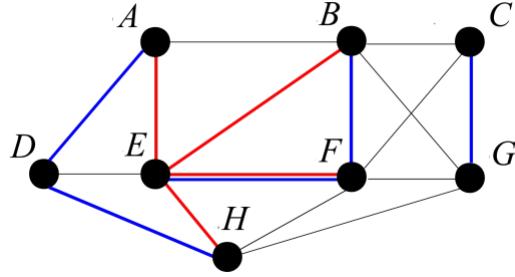


Figura 8.6: Dimostrazione dell'assioma di scambio

8.11.1 Matroide uniforme per KP/FKP

$$\S \subseteq \mathcal{B} : |\S| \leq \lfloor \frac{\mathcal{V}}{2} \rfloor$$

- Assioma banale: l'insieme vuoto rispetta il vincolo di cardinalità.
- Assioma di eredità: se x rispetta il vincolo di cardinalità, tutti i suoi sottoinsiemi lo rispetteranno.
- Assioma di scambio: se x e y rispettano il vincolo di cardinalità e $|x| = |y| + 1$, uno può sempre aggiungere un elemento adatto di x ad y senza violare la cardinalità.

Questo per quanto riguarda FKP, per il KP i primi due assiomi vengono mantenuti, mentre l'ultimo no. Consideriamo il seguente esempio:

$$V = 6$$

$$v = [33221]$$

Sia $x = \{3, 4, 5\}$ che $y = \{1, 2\}$ si trovano in \mathcal{F} , ma nessun elemento di x può essere aggiunto ad y .

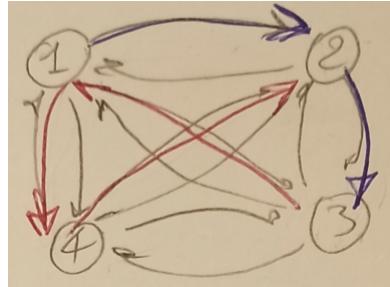


Figura 8.7: Esempio TSP

8.11.2 Matroide grafico MST (Minimum Spanning Tree)

$$\mathcal{F} = \{x \subseteq B : x \text{ non forma cicli}\}$$

- Assioma banale: l'insieme vuoto degli archi non forma cicli.
- Assioma di eredità: se x è aciclico, tutti i suoi sottoinsiemi sono aciclici.
- Assioma di scambio: se x ed y sono aciclici e $|x| = |y| + 1$, allora è sempre possibile aggiungere un arco "adatto" (che rispetta il vincolo) di x ad y senza formare alcun ciclo.

8.11.3 Matroide per TSP

Nel caso del TSP i primi due assiomi vengono mantenuti:

- l'insieme vuoto non ha percorsi ed gradi sono ≤ 1
- un qualsiasi sottoinsieme proprio di un insieme $\in \mathcal{F}$ appartiene ad \mathcal{F} .

Tuttavia il terzo assioma viene violato, consideriamo il seguente esempio: $y = \{(1,2), (2,3)\}$ ed $x = \{(3,1), (1,4), (4,2)\}$.

L'ottimalità dell'algoritmo Greedy può essere dimostrata utilizzando i gree-doidi, se l'assioma di scambio viene rafforzato:

$$\begin{cases} x \in \mathcal{F}, y \in \mathcal{B}_{\mathcal{F}} \text{ tale che } x \subseteq y \\ i \in B \setminus y \text{ tale che } x \cup \{i\} \in \mathcal{F} \end{cases} \implies \exists j \in y \setminus x : \begin{cases} x \cup \{j\} \in \mathcal{F} \\ y \cup \{i\} \setminus \{j\} \in \mathcal{F} \end{cases}$$

Data una base ed uno dei suoi sottoinsiemi (dalla quale la base risulta accessibile), se è presente un elemento che "si smarrisce" nel sottoinsieme della base, deve esserci un altro elemento che continua sul percorso giusto e deve essere fattibile scambiare due elementi nella base.

8 EURISTICHE COSTRUTTIVE

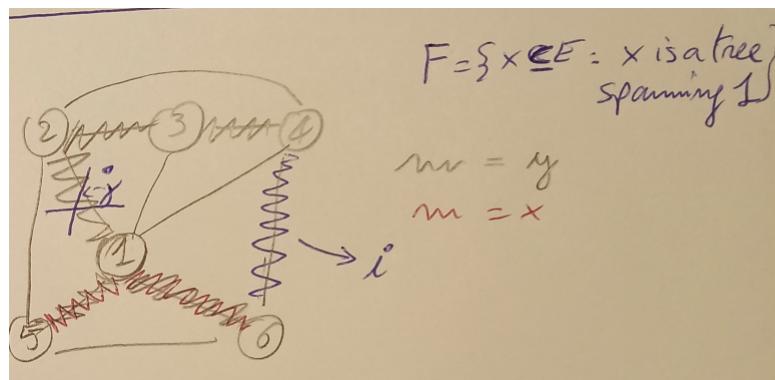
8.11.4 Greedoidi con forti assiomi di scambio : MST

Un esempio classi di greedoide con un forte assioma di scambio è dato da:

$B =$ insieme degli archi di un grafo

\mathcal{F} = collezione degli alberi che includono un dato vertice v_1

Questo porta all'algoritmo di Prim per il problema del minimo albero ricoprente



L'assioma banale e quello di accessibilità rimangono validi mentre (quello di eredità no). Mentre l'assioma di scambio mantiene una forma forte.

Notare che l'ottimalità di un algoritmo costruttivo A dipende da:

- le proprietà del problema (es. la funzione obiettivo additiva, basi come soluzioni fattibili).
 - le proprietà dello spazio di ricerca \mathcal{F}_A

8.12 Algoritmi non esatti

Quando le proprietà algebriche di un problema non vengono soddisfatte, tali che l'algoritmo e lo spazio di ricerca non compongano una matroide o un greedoide con un forte assioma di scambio, può essere fatta una cosa, ovvero cercare di guidare il comportamento dell'algoritmo all'interno dello spazio di ricerca utilizzando un *raffinamento* del criterio di selezione. Questo in maniera di farlo dipendere sia su i che su x , così da usare $\varphi_A(i, x)$.

Questo magari non dimostra che l'algoritmo raggiunga la soluzione ottima ma abbastanza spesso otterrà risultati migliori, e magari alcune approssimazione potranno essere provate.

Algoritmo euristico costruttivo per KP

Nel KP i tre assiomi non sono soddisfatti, poiché l'assioma di scambio non è

8 EURISTICHE COSTRUTTIVE

soddisfatto. In particolare, il problema è che l'oggetto non ha lo stesso volume ed una piccola cardinalità.

Se il problema non ammette uno spazio di ricerca con delle proprietà adatte, uno deve mantenere da conto i vincoli del problema adottando:

1. non solo una buona definizione di \mathcal{F}_A
2. ma anche una definizione sofisticata del criterio di selezione $\varphi_A(i, x)$, questo permette risultati efficaci, anche se non probabilmente ottimali.

Nel KP, l'inconveniente deriva dal volume degli oggetti: premettendo che gli oggetti abbiano un grande volume, ma anche un piccolo volume. Definiamo il criterio di selezione come il valore unitario $\varphi_A(i, x) = \frac{\varphi_i}{v_i}$. L'algoritmo risultante può avere delle brutte prestazioni, e con una piccola modifica può diventare 2-approximato.

Consideriamo il seguente esempio con uno zaino di volume $V = 8$:

B	a	b	c	d	e	f
φ	7	2	4	5	4	1
v	5	3	2	3	1	1
φ/v	1.4	0.67	2	1.67	4	1

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset, V_r = 8$
2. $x := \{e\}, V_r = 7$
3. $x := \{c, e\}, V_r = 5$
4. $x := \{c, d, e\}, V_r = 2$
5. $x := \{c, d, e, f\}, V_r = 1$, l'oggetto *a* non sta nello zaino.
6. Visto che $\Delta_A^+(x) = \emptyset$, termina.

Il valore della soluzione trovata è 14, mentre la soluzione ottima è $x^* = \{a, c, e\}$ ed il suo valore è 15 - l'algoritmo non ha delle brutte prestazioni in questo caso. Tuttavia, ci sono dei casi critici, come il seguente zaino con un volume $V = 10$. Questo è un algoritmo euristico, è buono? solitamente sì, ma sono presenti dei casi patologici dove non lo è:

B	a	b
φ	10	90
v	1	10
φ/v	10	9

L'algoritmo scegli l'oggetto *a*, il quale ha una rapporto più alto, quindi termina. La soluzione ha un valore 10 (con l'euristica), ma la soluzione ottima è 90. Il motivo di questo errore è dato dal primo oggetto scartato ha sia una grande volume ma anche un grande valore. Stiamo perdendo molto in questo caso. Vediamo un modo per risolvere questo problema:

8 EURISTICHE COSTRUTTIVE

Esempio di un'algoritmo 2-approssimato per KP Verrà mostrata una piccola dimostrazione di come questo algoritmo viene approssimato, tuttavia la dimostrazione all'esame non verrà chiesta, ma rimane come requisito sapere l'esistenza di un algoritmo 2-approssimato che può risolvere questo problema.

1. Si inizia con un sottoinsieme vuoto $x^{(0)} = \emptyset$
2. Trovare l'oggetto $i^{(t)}$ di valore unitario massimo in $B \setminus x^{(t-1)}$

$$i^{(t)} := \arg \max_{i \in B \setminus x^{(t-1)}} \frac{\phi_{i^{(t)}}}{v_{i^{(t)}}}$$

3. Nel caso in cui rispetti la capacità, aggiungere $i^{(t)}$ a $x^{(t-1)}$: $x^{(t)} := x^{(t-1)} \cup \{i^{(t)}\}$ e ritorna al passo 2.
4. Costruisci una soluzione con il primo oggetto rifiutato $x' = \{i^{(t)}\}$
5. Restituire la soluzione migliore tra x e x' : $f_A = \max[f(x), f(x')]$

Si può dimostrare facilmente che, la somma dei valori delle due soluzioni è una sovrastima della soluzione ottima

$$f(x) + f(x') = \sum_{\tau=1}^t \phi_{i(\tau)} \geq f^*$$

e che la migliore delle due soluzioni è almeno metà della loro somma

$$f_A = \max[f(x), f'(x)] \geq \frac{f(x) + f(x')}{2} \geq \frac{1}{2} f^*$$

Euristica Nearest Neighbour per il TSP

Consideriamo il problema del TSP su un grafo completo $G = (N, A)$ con il solito spazio di ricerca (sottoinsieme degli archi senza sotto percorsi e con un grado ≤ 1 su tutti i nodi).

L'algoritmo costruttivo trova sempre una soluzione fattibile, però la soluzione può essere infinitamente pessima.

Cambiando lo spazio di ricerca \mathcal{F}_A , in maniera che includa tutti cammini che iniziano dal nodo 1.

Sia N_x l'insieme dei nodi visitati da x : le estensioni accettabili di questi nodi sono tutti gli archi uscenti dall'ultimo nodo del cammino x e che non terminano in un sotto-cammino

$$\Delta_A^+(x) = \{(h, k) \in A : h = \text{Last}(x), k \notin N_x \text{ or } k = 1 \text{ and } N_x = N\}$$

Quindi un'estensione dell'attuale sotto insieme proposto come soluzione consiste in un arco tale che il nodo k non sia un nodo appartenente ad uno dei nodi già visitati a meno che non sia il primo nodo (quando tutti i nodi sono stati visitati $N_x = N$).

Controlliamo gli assiomi:

8 EURISTICHE COSTRUTTIVE

- l'assioma banale è mantenuto.
- l'assioma di accessibilità è mantenuto : poiché rimuovere un arco restituisce un cammino che inizia dal nodo 1 (rimane sempre un arco, ovvero una soluzione appartenente allo spazio di ricerca, secondo gli assiomi dei greedoidi).
- l'assioma di eredità non viene mantenuto: non tutti i sottoinsiemi sono cammini.
- l'assioma di scambio non viene mantenuto.

L'euristica NN (Nearest Neighbour) adotta la funzione obiettivo come criterio di selezione:

- Inizia con l'insieme vuoto degli archi $x^{(0)} = \emptyset$, che rappresenta un percorso degenere che esce dal nodo 1 (la soluzione ottima contiene sicuramente il nodo 1).
- Trovare l'arco di costo minimo uscente dall'ultimo nodo di x

$$(i, j) = \arg \min_{(h,k) \in \Delta_A^+(x)} c_{hk}$$

- Se $j \neq 1$, ritorna al punto 2; altrimenti, termina.

L'algoritmo è molto intuitivo e la sua complessità è $\Theta(n^2)$, non è esatto ma $\log n$ -approssimato.

Procediamo con un esempio numerico, consideriamo il seguente grafo completo:



Figura 8.8: Grafo completo (gli archi non sono riportati per maggiore chiarezza)

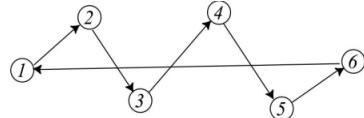


Figura 8.9: Euristica NN partendo dal nodo 1

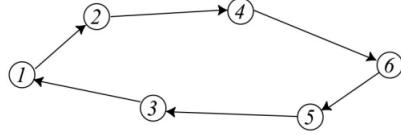


Figura 8.11: Effettiva soluzione ottima

Eseguendo l'algoritmo partendo dal nodo 1 otteniamo il percorso sotto-ottimale mostrato in figura 8.9. Tuttavia avviare l'algoritmo dal nodo 2 restituisce anch'esso un percorso sotto ottimale.

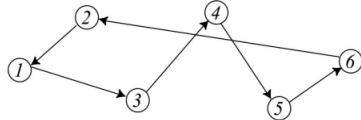


Figura 8.10: Euristica NN partendo dal nodo 2

Più precisamente, avviare l'esecuzione dell'euristica da un qualsiasi nodo restituirà una soluzione sotto ottimale, significa che la soluzione ottima non è restituita da questo algoritmo dato un qualsiasi nodo.

8.13 Algoritmi costruttivi puri ed adattivi

Un algoritmo costruttivo A è:

- **puro** se il criterio di selezione φ_A dipende solo dal nuovo elemento i .
- **adattivo** se φ_A dipende sia da i che dalla soluzione corrente x .

Molto criteri $\varphi_A(i, x)$ ammettono forme equivalenti dipendendo solamente da i :

- nel KP, $\varphi_A(i, x) = f(x \cup \{i\})$ è equivalente a ϕ_i (questo perché la funzione obiettivo è additiva, si può concentrarsi quindi solo sul valore ϕ).
- nel TSP, $\varphi_A((i, j), x) = f(x \cup \{(i, j)\})$ è equivalente a c_{ij}

Fin'ora abbiamo solamente visto algoritmi puramente costruttivi. Un criterio di selezione additivo restituisce un algoritmo puramente costruttivo.

Mentre per dimostrazione la definizione di *adattivo* utilizziamo il problema del Set Covering.

Set Covering

Data una matrice binaria ed un vettore dei costi associato alle colonne, si vuole trovare un sottoinsieme delle colonne di costo minimo tali che coprano tutte le righe.

La funzione obiettivo è additiva, ma le soluzioni non sono delle basi, in quanto non sono sottoinsiemi massimali, questo perché più piccola la soluzione è meglio risulta. Inoltre, risulta necessario un criterio di selezione adattivo $\varphi_A(i, x)$, poiché uno *puro* $\varphi_A(i)$ potrebbe ripetutivamente scegliere colonne che coprono le stesse righe.

Le idee più promettenti da seguire sono:

- la **funzione obiettivo** seleziona colonne di costo minimo.
- i vincoli: selezionare colonne che coprono molte righe.
- il corrente sottoinsieme x : selezionare colonne che coprono nuove righe.

Ricapitolando:

- includere nel passo $\Delta_A^+(x)$ solo colonne che coprono righe addizionali non presenti in x .
- applicare il criterio di selezione adattivo $\varphi_A(i, x) = \frac{c_i}{a_i(x)}$ dove $a_i(x)$ è il numero di righe coperte da i , ma che non sono presenti in x .

Consideriamo il seguente esempio positivo:

c	3	5	6	2	1	7	1	8
A	1	0	1	0	0	0	0	1
	0	1	0	0	0	1	0	0
	0	0	0	1	0	0	0	0
	1	0	1	0	0	1	0	0
	0	0	0	0	1	0	1	1
	1	0	0	1	0	1	0	0

L'algoritmo esegue i seguenti passi:

1. $x := \emptyset$;
2. selezionare $i := 1$ ed aggiorna $x := \{1\}$;
3. selezionare $i := 5$ ed aggiorna $x := \{1, 5\}$;
4. selezionare $i := 4$ ed aggiorna $x := \{1, 4, 5\}$;
5. selezionare $i := 2$ ed aggiorna $x := \{1, 2, 4, 5\}$;
6. tutte le righe sono coperte, quindi $\Delta_A^+(x) = \emptyset$ e termina.

8 EURISTICHE COSTRUTTIVE

Il valore trovato è 11 ed è l'ottimo. Ma non è sempre così, guardiamo un conto esempio. Consideriamo il seguente esempio negativo:

c	25	6	8	24	12
A	1	1	0	0	0
	1	1	0	0	0
	1	1	1	0	0
	1	0	1	1	0
	1	0	0	1	0
	1	0	0	0	1

Passi dell'algoritmo:

1. $x := \emptyset$;
2. Visto che $\frac{c_i}{a_i}(x) = [4.16 \ 2 \ 4 \ 12 \ 12]$ allora selezionare $i := 2$,
3. Visto che $\frac{c_i}{a_i}(x) = [8.3 - 8 \ 12 \ 12]$ allora selezionare $i := 3$,
4. Visto che $\frac{c_i}{a_i}(x) = [12.5 - -24 \ 12]$ allora selezionare $i := 5$,
5. Visto che $\frac{c_i}{a_i}(x) = [25 - -24 -]$ allora selezionare $i := 2$,
6. tutte le righe sono coperte, $\Delta_A^+(x) = \emptyset$ e termina.

La soluzione restituita è $x = \{2, 3, 4, 5\}$ ed il suo valore è 50, dove la soluzione ottimale $x^* = \{1\}$ ha valore $f^* = 25$.

Approssimabilità del SCP

Questo algoritmo ha un rateo di approssimazione logaritmico e non-costante.

- ad ogni passo t , ogni colonna i viene valutata con criterio

$$\varphi_A(i, x^{(t-1)}) = \frac{c_i}{a_i(x^{(t-1)})}$$

- ogni riga j è coperta da una certa colonna i_j ad un dato passo t_j
- quando ogni riga j viene coperta, il suo peso viene impostato a:

$$\theta_j = \frac{c_{i_j}}{a_{i_j}(x^{(t_j-1)})}$$

quindi il peso totale delle righe incrementa di c_{i_j} al passo t_j ; corrispettivamente, x include la colonna i_j ed il suo costo incrementa di c_{i_j} .

- il costo totale di x è sempre uguale al peso totale delle righe

$$f_A(x) = \sum_{i \in x} c_i = \sum_{j \in R} \theta_j$$

8.14 Euristica First-Fit

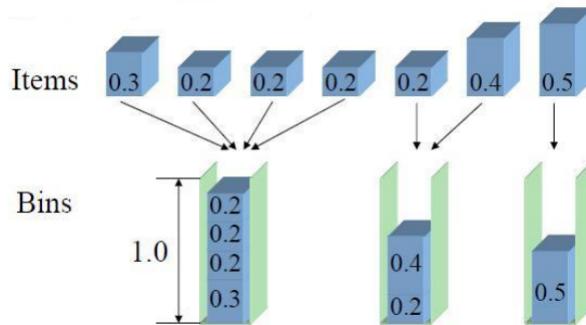
Bin Packing Problem

Il BPP richiede di dividere un insieme E di oggetti voluminosi in un minimo numero di container di una data capacità presi da un insieme C . Il ground set $B = E \times C$ include gli assegnamenti a coppie di oggetti-container (i, j) , con esattamente un container per ogni oggetto, con esattamente il volume totale in ogni container che non eccede la capacità.

Definiamo lo spazio di ricerca \mathcal{F}_A come l'insieme di tutte le soluzioni parziali. La funzione obiettivo è un pessimo criterio di selezione, perché è *piatta* (tutti i sottoinsiemi aumentati hanno lo stesso valore o sono incrementati di 1).

First-Fit

- Si inizia con un insieme vuoto $x^{(0)} = \emptyset$
- Seleziona la coppia (i, j) in accordo al seguente criterio:
 - i è il primo oggetto non assegnato.
 - j è il primo container utilizzato con abbastanza capacità residua per i , o se nessun container non ha abbastanza capacità residua, il primo container *inutilizzato*.
- aggiungere un nuovo assegnamento alla soluzione $x^{(t)} := x^{(t-1)} \cup \{(i, j)\}$



Notare che la scelta (i, j) dipende sia da i che da x , mentre non è determinata da $f(x \cup \{(i, j)\})$.

La soluzione non è ottima:

ma appunto approssimata (dimostrazione, come sempre non sono chieste all'esame):

8 EURISTICHE COSTRUTTIVE

- sono necessari almeno $f^* \geq \sum_{i \in E} \frac{v_i}{V}$ container.
- il volume occupato è $> \frac{V}{3}$ per tutti i container utilizzati, possibilmente eccetto per l'ultimo.
- il volume totale eccede quello di $f_A - 1$ container "saturi"

$$\sum_{i \in E} v_i > (f_A - 1) \frac{V}{2}$$

cosa che implica $(f_A - 1) \leq 2 \frac{\sum_{i \in E} v_i}{V} \leq 2f^* \implies f_A \leq 2f^* + 1$

Euristica First-Fit con decremento

Il rateo di approssimazione $\alpha = 2$ viene mantenuto per qualsiasi permutazione degli oggetti.

L'intuizione suggerirebbe di selezionare prima gli oggetti più piccoli, questo per fare in modo di mantenere la funzione obiettivo $f(x \cup \{i\})$ più piccola possibile, ma questo trascura il fatto che tutti gli oggetti debbano essere assegnati (intuizione stupida).

Per contrasto, è meglio selezionare l'oggetto più grande prima perché:

- ogni oggetto in un container ha un volume strettamente più grande della capacità residua di tutti i container precedenti.
- mantenere gli oggetti più piccoli alla fine garantisce che differenti container abbiano una capacità residua piccola abbastanza per contenerli.

Questo algoritmo ha un rateo di approssimazione migliore $f_A \leq \frac{11}{9}f^* + 1$

8.15 Estensione delle euristiche costruttive

Lo schema base degli algoritmi costruttivi può essere potenziato utilizzando:

1. un grafo di costruzione più efficace:
 - aggiungi più di un elemento al corrente sottoinsieme x .
 - aggiungi elementi ad x , ma anche rimuovi gli elementi da x .
2. un criterio di selezione più sofisticato, come:
 - una funzione **regret-based** ("basata sul rimpianto"), che valuta potenziali perdite future associate con l'elemento i .
 - una funzione **look-ahead** che valuta il valore finale dell'obiettivo ottenuto aggiungendo i ad x .

L'algoritmo costruttivo aggiunge un elemento alla volta alla soluzione, è possibile generalizzare questo schema con due possibili varianti che ad ogni passo:

8 EURISTICHE COSTRUTTIVE

1. aggiungono più di un elemento: il criterio di selezione $\varphi_A(B^+, x)$ il nuovo parametro identifica un sottoinsieme $B^+ \subseteq B \setminus x$ da aggiungere, anziché un singolo elemento i .
2. aggiungere elementi, ma anche rimuovere un numero più piccolo di elementi: il criterio di selezione $\varphi_A(B^+, B^-, x)$ identifica un sottoinsieme $B^+ \subseteq B \setminus x$ da aggiungere ed un sottoinsieme $B^- \subseteq x$ da rimuovere con $|B^+| > |B^-|$.

Questi algoritmi costruiscono un grafo di costruzione aciclico sullo spazio di ricerca, così che loro non rivisiteranno mai un qualsiasi sottoinsieme.

Il problema fondamentale è che definire famiglie di sottoinsieme tali che ottimizzare il criterio di selezione sia un problema polinomiale.

$$\min_{B^+ \subseteq B \setminus \{x\}, B^- \subseteq \{x\}} \varphi(B^+, B^-, x)$$

8.15.1 Euristica di distanza per Stainer Tree Problem (STP)

Dato un grafo indiretto $G = (V, E)$ ed una funzione di costo $c : E \rightarrow \mathbb{N}$ definita su gli archi ed un sottoinsieme di vertici speciali $U \subset V$, trovare un albero di costo minimo che sia in grado di connettere tutti i vertici speciali.

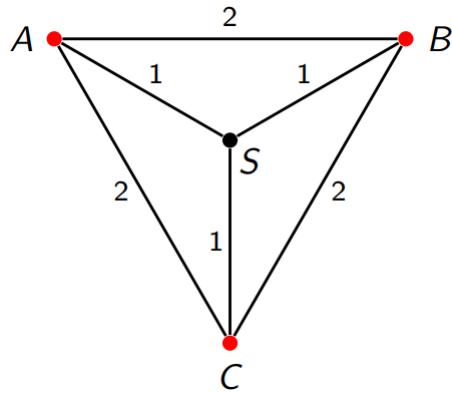


Figura 8.12: Albero del problema STP

Il minimo albero ricoprente che copre tutti i vertici speciali non è necessariamente ottimale (e potrebbe non esistere).

Distance heuristic per il STP

Un algoritmo costruttivo basilare potrebbe adottare gli stessi spazi di ricerca dell'algoritmo di Kruskal (insieme di tutte le foreste) o Prim (insieme di tutti gli alberi che includono un vertice speciale) per trovare un MST/MAR.

8 EURISTICHE COSTRUTTIVE

Questo però aggiungendo un arco alla volta, mi porta nel primo caso a restituire soluzioni con archi ridondanti, e nel secondo algoritmo abbiamo una difficoltà nel distinguere gli archi utili da quelli ridondanti.

L'euristica di distanza adotta come spazio di ricerca \mathcal{F} la collezione di tutti gli alberi inclusi un dato vertice speciale v_1 (come per Prim). Iterativamente aggiunge un percorso B^+ tra x e un vertice speciale anziché un singolo arco, così che:

- x rimane un albero
- x copre un nuovo vertice speciale
- il cammino di costo minimo può essere calcolato in maniera efficiente ad ogni passo.

Esso termina quando tutti i vertici speciali sono connessi.

Esempio

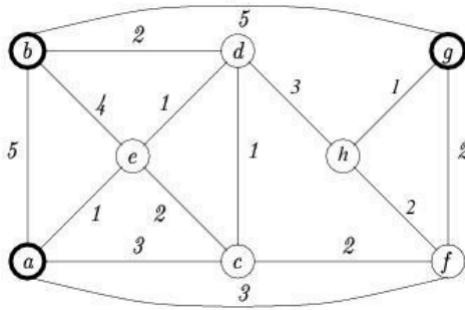


Figura 8.13: Passo 1

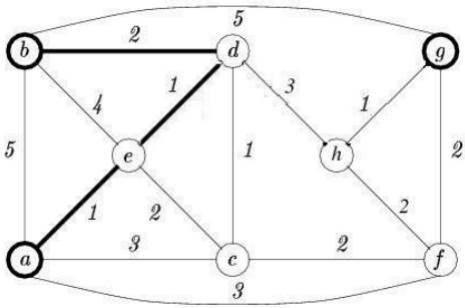


Figura 8.14: Passo 2

8 EURISTICHE COSTRUTTIVE

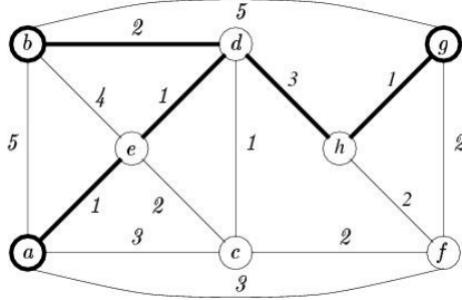


Figura 8.15: Passo 3

1. Si inizia con un singolo vertice speciale a , e il sottoinsieme delle soluzione è un albero degenere $x := \emptyset$ (figura 8.13).
2. Si aggiunge il vertice speciale più vicino b attraverso il cammino $\langle a, e, d, b \rangle$, $x = \{(a, e), (e, d), (d, b)\}$ (figura 8.14).
3. si aggiunge il vertice speciale più vicino g attraverso il cammino $\langle g, h, d \rangle$, $x = \{(a, e), (e, d), (d, b), (g, h), (h, d)\}$ (figura 8.15).
4. tutti i vertici speciali si trovano nella soluzione, termina.

L'algoritmo della distanza euristica risulta 2-approximato, è equivalente a calcolare li minimo albero ricoprente su un grafo con i vertici ridotti ai vertici speciali, e gli archi corrispondenti ai cammini minimi.

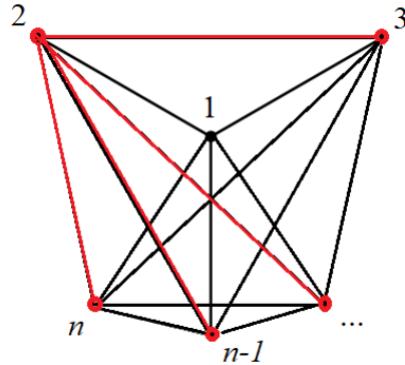
Controesempio dell'ottimalità

Consideriamo un grafo completo $G = (V, E)$ con $U = V \setminus \{1\}$ e costo

$$c_{uv} = \begin{cases} (1 + \epsilon)M & \text{per } u \text{ o } v = 1 \\ 2M & \text{per } u, v \in U \end{cases}$$

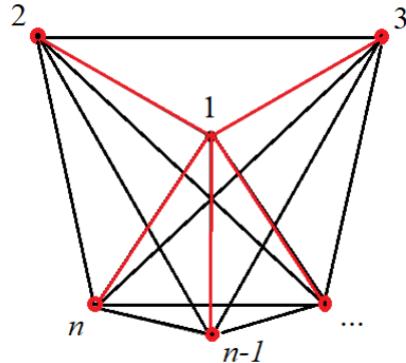
L'euristica di distanza restituisce una grafo a stella che copre i vertici speciali:

$$f_{DH} = (n - 2) \cdot 2M$$



la soluzione ottimale risulta una stella che è centrata rispetto al nodo 1:

$$f^* = (n - 1) \cdot (1 + \epsilon)M$$

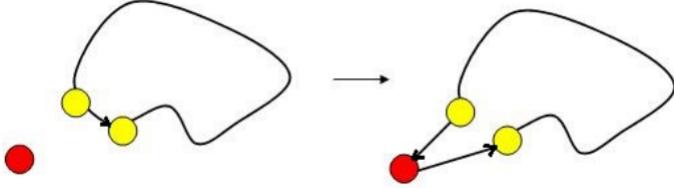


Il rateo di approssimazione è $\rho_{DH} = \frac{f_{DH}}{f^*} = \frac{n-2}{n-1} \cdot \frac{2}{1+\epsilon} < 2$ e converge verso 2 all'aumentare di n ed al decrementare di ϵ .

8.15.2 Algoritmi di inserzione per il TSP

Differenti algoritmi euristici per il TSP definiscono lo spazio di ricerca \mathcal{F}_A come l'insieme di tutti i cicli del grafo rispetto ad un dato nodo. Un ciclo non può essere ottenuto da un altro nodo aggiungendo un singolo arco, invece può essere ottenuto aggiungendo due archi $(i, k), (k, j)$ e rimuovendone uno (i, j) .

8 EURISTICHE COSTRUTTIVE



1. Iniziare con un ciclo sul nodo stesso (ricorsivo) di costo zero: $x^{(0)} = \{(1, 1)\}$, possiamo notare che non è molto differente da un insieme vuoto.
2. Selezionare un nodo k da aggiungere ed un arco (i, j) da rimuovere.
3. Se il ciclo non visita tutti i nodi, allora tornare al punto 2; altrimenti termina.

Tale schema non visita mai la stessa soluzione e costruisce una soluzione fattibile in $n - 1$ passi, dove ad ogni passo si aggiunge un nuovo nodo.

Il criterio di selezione $\varphi_A(B^+, B^-, x)$ deve scegliere un arco ed un nodo; ci sono $(n - |x|)|x| \in O(n^2)$ alternative possibili:

- $|x|$ possibili archi (s_i, s_{i+1}) da rimuovere.
- $n - |x|$ possibili nodi k da aggiungere attraverso gli archi (s_i, k) e (k, s_{i+1}) .

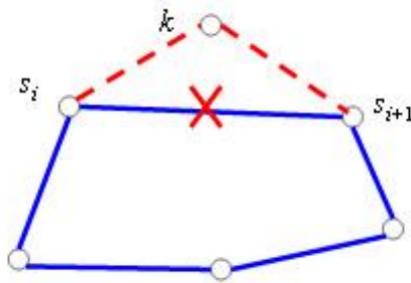
L'euristica con **inserzione economica** (CI, *Cheapest Insertion*) utilizza un criterio di selezione:

$$\varphi_A(B^+, B^-, x) = f(x \cup B^+ \setminus B^-)$$

La funzione obiettivo $f(x)$ è additiva, quindi estensibile all'intero spazio di ricerca \mathcal{F}_A , visto che $f(x \cup B^+ \setminus B^-) = f(x) + c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}}$

$$\arg \min_{B^+, B^-} \varphi_A(B^+, B^-, x) = \operatorname{argmin}_{i, k} (c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$$

Il costo computazionale di calcolare φ_A decrementa da $\Theta(n)$ a $\Theta(1)$.



Algoritmo Cheapest Insertion

- Inizia con auto-ciclo di costo zero sul nodo 1: $x^{(0)} = \{(1, 1)\}$, è come partire con un singolo nodo.
- Selezione dell'arco $(s_i, s_{i+1}) \in x$ ed il nodo $k \notin N_x$ tale che $(c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$ sia il minimo.
- Se il ciclo non visita tutti i nodi, allora torna al punto 2; altrimenti termina.

Non è un algoritmo esatto, però è 2-approssimato, sotto la disuguaglianza triangolare. Consideriamo il seguente esempio:

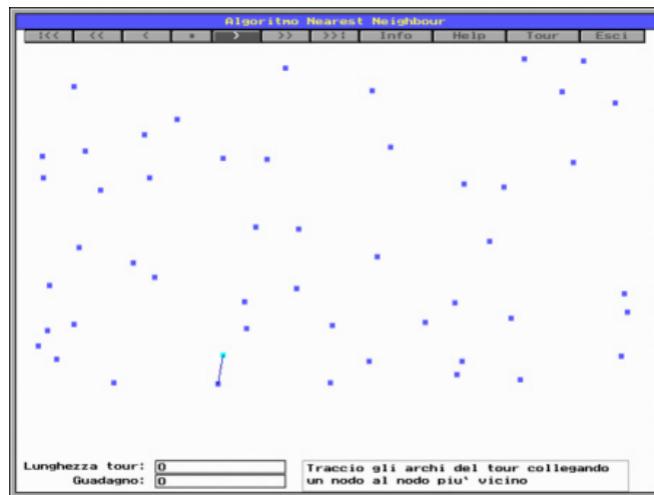


Figura 8.16: Si inizia con un singolo nodo (come nell'euristica NN)

8 EURISTICHE COSTRUTTIVE

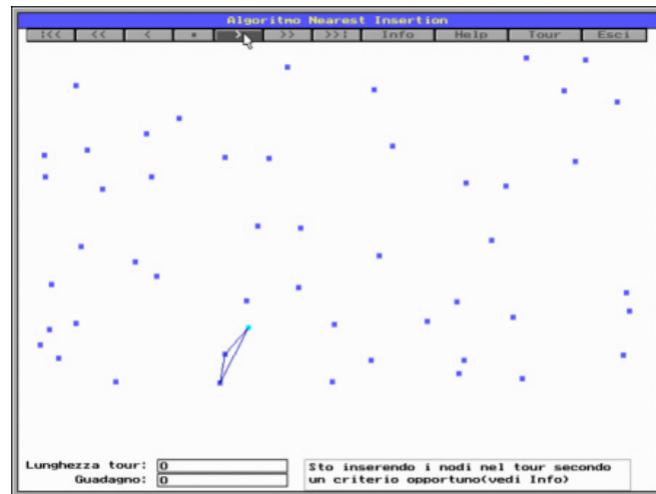


Figura 8.17: Si crea un ciclo (anziché un cammino)

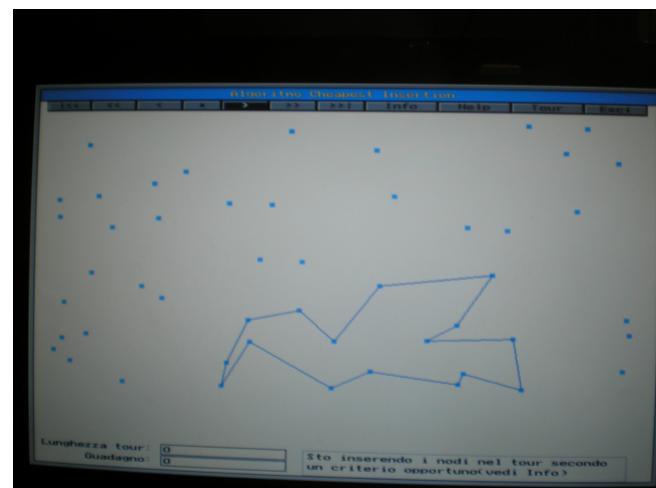


Figura 8.18: Si aggiunge ad ogni passo il nodo che minimamente incrementa il costo del circuito

8 EURISTICHE COSTRUTTIVE

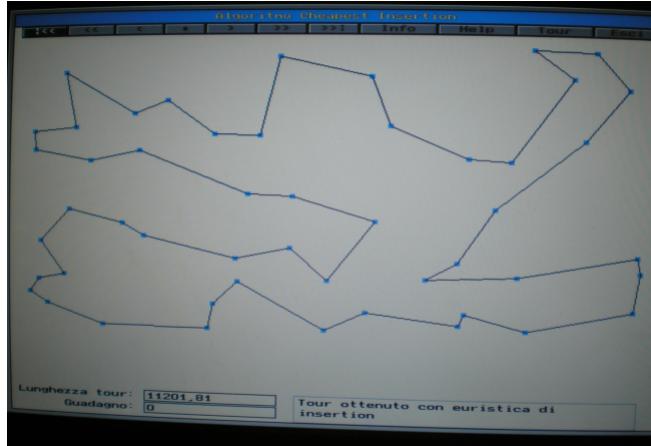


Figura 8.19: Si termina quando il circuito visita tutti i nodi

L'algoritmo CI effettua $n - 1$ passi, ad ogni passo t :

- calcola $(n - t)t$ coppie nodo-arco. Ogni valutazione richiede costo costante, e possibilmente aggiornare la mossa migliore.
- esegue la migliore aggiunta/rimozione.
- decide quando terminare.

La complessità totale è $\Theta(n^3)$, può essere ridotto a $\Theta(n^2 \log n)$ mettendo all'interno di un min-heap il migliore costo di inserzione per ogni nodo esterno, ognuno degli n passi:

- seleziona la migliore inserzione in un tempo $O(1)$
- viene eseguito creando due nuove inserzioni per ogni nodo esterno, la migliore delle quali possibilmente migliora il valore salvato nello heap; ognuno dei $O(n)$ miglioramenti impiega $O(\log n)$.

Euristica Nearest Insertion per il TSP

L'algoritmo CI tende a selezionare i nodi chiusi nel circuito x : minimizzando $c_{s_i,k} + c_{k,s_{i+1}} - c_{s_i,s_{i+1}}$ implica che $c_{s_i,k}$ e $c_{s_{i+1},k}$ sono piccoli. Per accelerare, uno può decomporre il criterio φ_A in due fasi. Algoritmo NI (*Nearest Insertion*):

1. Si con un auto-ciclo di costo zero sul nodo 1: $x^{(0)} = \{(1, 1)\}$.
2. **Criterio di aggiunta:** selezionare il nodo k più vicino al circuito x

$$k = \arg \min_{l \notin N_x} \left(\min_{s_i \in N_x} c_{s_i, l} \right)$$

8 EURISTICHE COSTRUTTIVE

3. **Criterio di rimozione:** selezionare l'arco s_i, s_{i+1} che minimizza f

$$(s_i, s_{i+1}) = \arg \min_{(s_i, s_{i+1}) \in x} (c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$$

4. Se il ciclo non visita tutti i nodi, allora tornare al punto 2; altrimenti termina.

Non è un algoritmo esatto ma è 2-approximato, sotto la diseguaglianza triangolare. Consideriamo il seguente esempio:

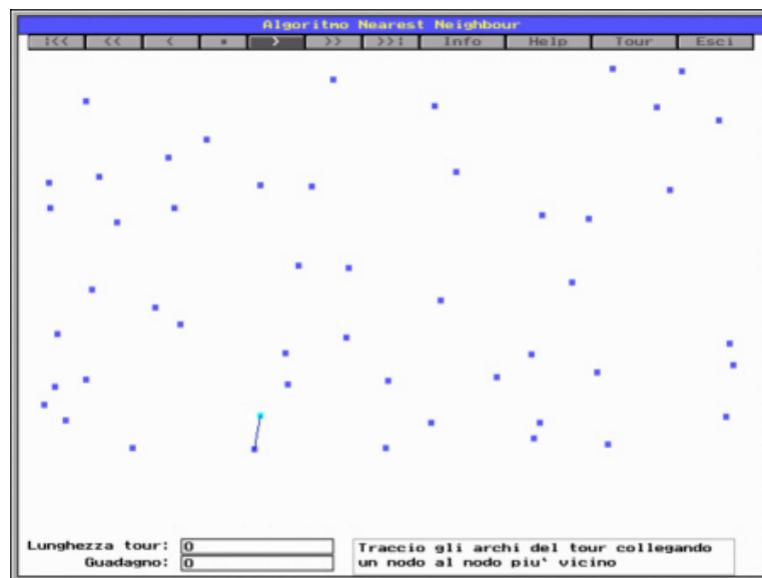


Figura 8.20: Si inizia con un singolo vertice (come per NN e CI)

8 EURISTICHE COSTRUTTIVE

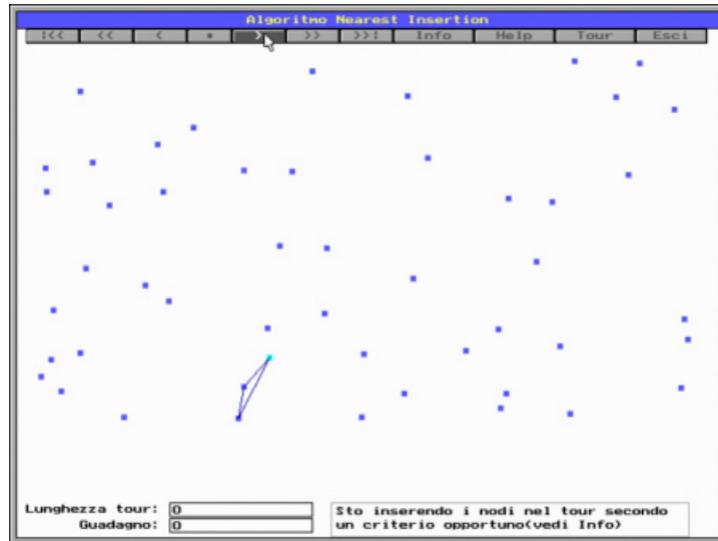


Figura 8.21: Si crea un ciclo (come nel CI)

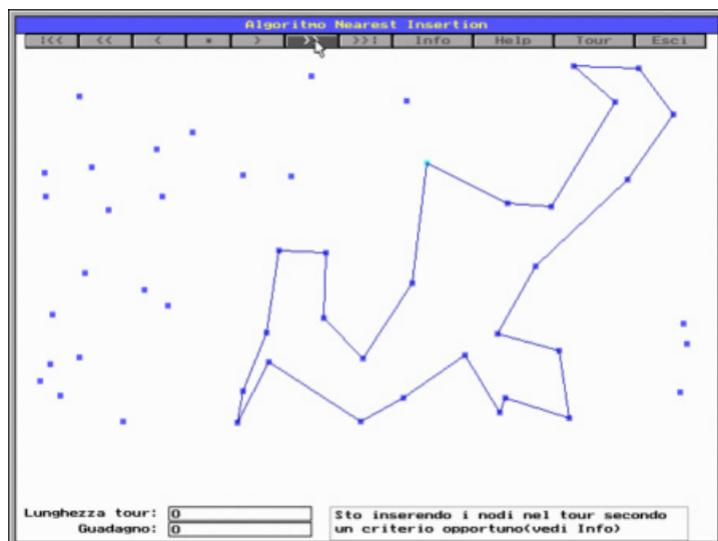


Figura 8.22: Il ciclo cresce differentemente, si aggiunge sempre il nodo più vicino, anche se questo incrementa il costo del ciclo più che aggiungere un altro nodo

8 EURISTICHE COSTRUTTIVE

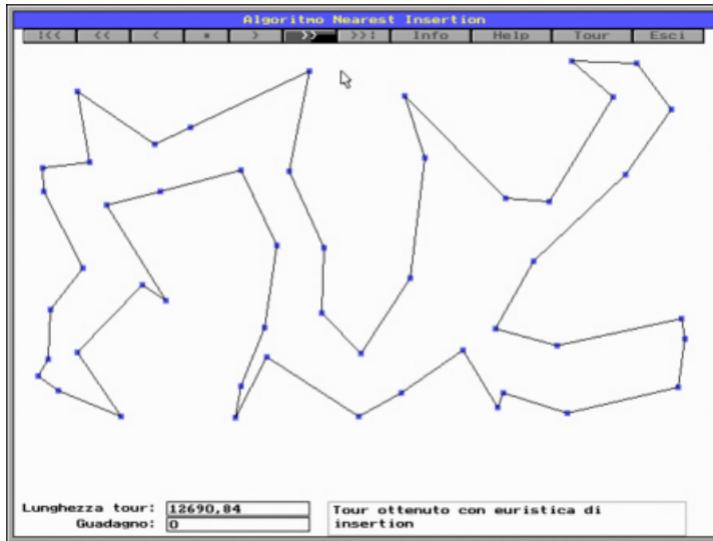


Figura 8.23: Si termina quando il ciclo ha visitato tutti i nodi

L'algoritmo NI effettua $n - 1$ passi, ad ogni passo t :

- calcola la distanza da $(n - t)$ nodi dal ciclo, ogni uno in $\Theta(t)$.
- Seleziona il nodo con distanza minima.
- Calcola la rimozione di t archi, ogni uno in tempo $\Theta(1)$
- Effettua la migliore addizione/rimozione
- Decide quando terminare

La complessità totale è $\Theta(n^3)$, può essere ridotta a $\Theta(n^2)$ utilizzando un vettore che contiene per ogni nodo esterno il suo nodo interno più vicino, ognuno dei $n - 1$ passi:

- seleziona il nodo più vicino in $O(n)$.
- crea due nuove inserzioni per ogni nodo esterno, la migliore delle quali possibilmente migliore il valore salvato nel vettore; ognuno dei $O(n)$ miglioramenti si prende $O(1)$ tempo.

Euristica Farthest Insertion per il TSP

La scelta del nodo più vicino al ciclo è naturale, ma ingannevole: visto che tutti i nodi devono essere visitati, è preferibile andare trattare per primi i nodi più problematici (*farthest*).

1. iniziare con un auto-ciclo a costo zero sul nodo 1: $x^{(0)} = \{(1, 1)\}$

8 EURISTICHE COSTRUTTIVE

2. criterio di aggiunta: selezionare il nodo k più lontano dal ciclo x

$$k = \arg \max_{l \notin N_x} \left(c_{s_i, l} \right)$$

3. criterio di rimozione: selezionare l'arco (s_i, s_{i+1}) che minimizza i costi

$$(s_i, s_{i+1}) = \arg \min_{(s_i, s_{i+1}) \in x} (c_{s_i, k} + c_{k, s_{i+1}} - c_{s_i, s_{i+1}})$$

4. se il ciclo non visita tutti i nodi, tornare al punto 2; altrimenti termina.

questa euristica risulta $\log n$ -approssimata sotto la disuguaglianza triangolare, quindi peggiore delle precedenti euristiche (caso peggiore).

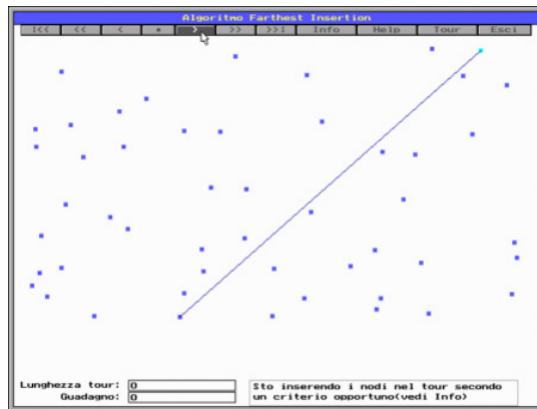


Figura 8.24: si raggiunge subito il primo nodo più lontano da quello iniziale

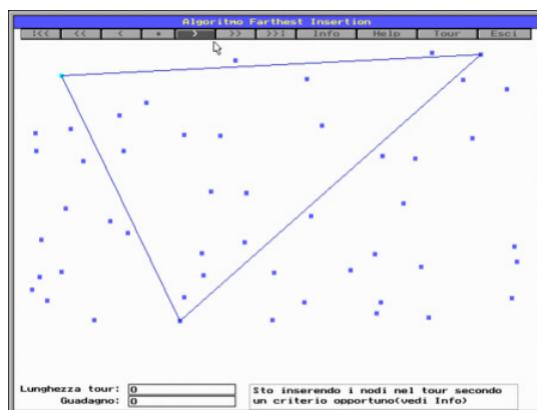


Figura 8.25: si prosegue nella stessa maniera

8 EURISTICHE COSTRUTTIVE

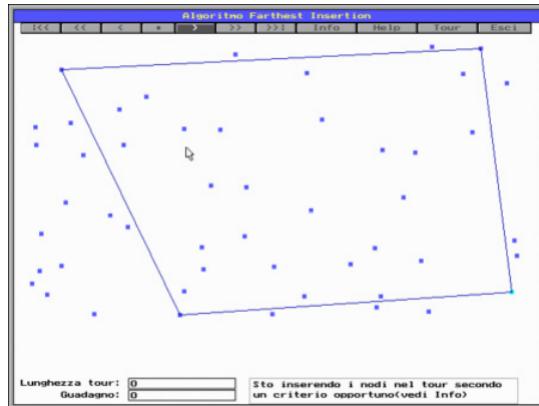


Figura 8.26: si cerca l'inserimento nella migliore maniera possibile

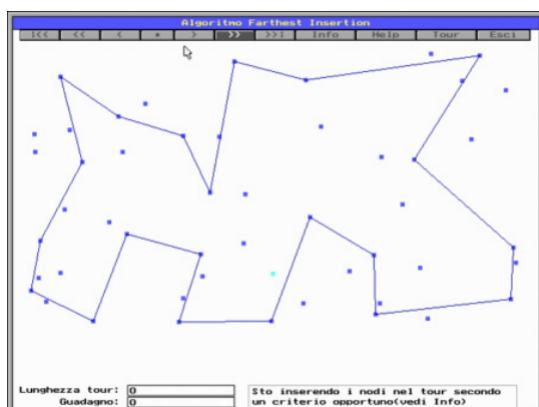


Figura 8.27: il ciclo cresce con più regolarità, senza attraversamenti

8 EURISTICHE COSTRUTTIVE

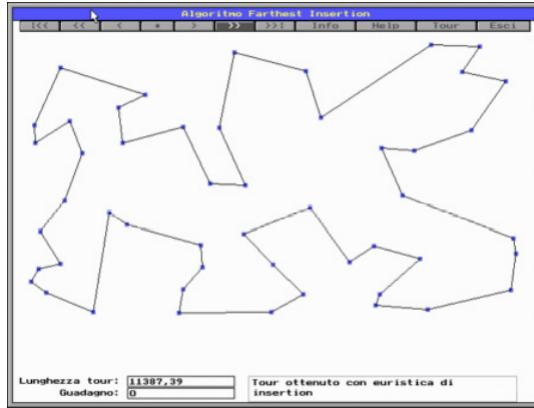


Figura 8.28: termina quando il ciclo visita tutti i nodi

L'algoritmo FI esegue in $n - 1$ passi, ad ogni passo t :

- calcola la distanza di $(n - t)$ nodi dal ciclo, ognuno in $\Theta(t)$ tempo.
- seleziona il nodo della distanza massima.
- calcola la rimozione di t archi, ognuno in tempo $\Theta(1)$.
- effettua la migliore rimozione/addizione.
- decide quando terminare.

La complessità generale è di $\Theta(n^3)$, può essere ridotto a $\Theta(n^2)$ come nell'eutistica NI.

8.15.3 Efficacia ed efficienza

Un algoritmo euristico costruttivo esegue al massimo $n = |B|$ passi che consistono in:

1. la costruzione $\Delta_A^+(x)$
2. il calcolo di $\varphi_A(i, x)$ per ogni $i \in \Delta_A^+(x)$
3. la selezione dell'elemento i che minimizza $\varphi_A(i, x)$
4. l'aggiornamento di x (e strutture dati ausiliarie)

In generale, la complessità è un polinomio di basso ordine dominato dalle prime due componenti citate.

$$T_A(n) \in O\left(n(T_{\Delta_A^+(n)} + T_{\varphi_A(n)})\right)$$

8 EURISTICHE COSTRUTTIVE

8.15.4 Caratteristiche generali degli algoritmi costruttivi

- sono intuitivi.
- sono semplici da progettare, analizzare ed implementare.
- sono molto efficienti.
- hanno un'efficacia fortemente variabile:
 - su alcuni problemi garantiscono una soluzione ottimale.
 - su altri problemi forniscono una garanzia di approssimazione.
 - sulla maggior parte dei problemi forniscono soluzioni di qualità estremamente variabile, solitamente scarsa.
 - su alcuni problemi non possono nemmeno garantire una soluzione fattibile.

Perciò risulta fondamentale studiare il problema prima dell'algoritmo.

8.15.5 Quando sono usati gli algoritmi costruttivi

- quando forniscono una soluzione ottimale.
- quando il tempo di esecuzione è molto corto.
- quanto il problema ha una dimensione rilevante o richiede computazioni complesse.
- come componente di altri algoritmi, per esempio come: fase iniziale (algoritmi di scambio), procedura base (algoritmi di ricombinazione).

8.15.6 Euristiche distruttive

Questo è un approccio esattamente complementare alleuristiche costruttive.

- si inizia a partire dall'intero ground set: $x^{(0)} := B$
- si rimuove un elemento per volta:
 - in maniera da rimanere nello spazio di ricerca \mathcal{F}_A .
 - massimizzando il criterio di selezione $\varphi_A(i, x)$
- termina con $\Delta_A^+(x) = \emptyset$ (non è possibile rimanere nello spazio di ricerca \mathcal{F}_A).

8 EURISTICHE COSTRUTTIVE

un euristica distruttiva può essere descritta come:

Algorithm 7 Euristica distruttiva - *Stingy(I)*

```

 $x := B$ 
 $x* := B$ 
if  $x \in X$  then
     $f^* := f(x)$ 
else
     $f^* := +\infty$ 
end if
while  $\Delta_A^+(x) \neq \emptyset$  do
     $i := \arg \max_{i \in \Delta_A^+(x)} \varphi_A(i, x);$ 
     $x := x \setminus \{i\};$ 
end while
if  $x \in X$  and  $f(x) < f^*$  then
     $x^* := x;$ 
     $f^* := f(x);$ 
end if
Return( $x^*, f^*$ );

```

Perché non sono molto utilizzate? Quando le soluzioni sono molto più piccole del ground set $|x| << |B|$ un'euristica distruttiva:

- richiede un maggior numero di passi.
- è più possibile che effettui una decisione erronea ad uno dei primi step.
- a volte richiede più tempo per calcolare $\Delta_A^+(x)$ e $\varphi_A(i, x)$

Quando una euristica costruttiva restituisce soluzioni ridondanti, è utile accordare una euristica distruttiva al suo termine come fase di post-processing.

L'euristica distruttiva ausiliaria:

- inizia da una soluzione x dell'euristica costruttiva, anziché B .
- adotta come spazio di ricerca la regione di fattibilità:

$$\mathcal{F}_A = X \implies \Delta_A^+(x) = \{i \in x : x \setminus \{i\} \in X\}$$

- adotta come criterio di selezione la funzione obiettivo:

$$\varphi_A(i, x) = f(x \setminus \{i\})$$

- termina dopo pochi passi.

8 EURISTICHE COSTRUTTIVE

Euristica Costruttiva/Distruttiva per il SCP

c	6	8	24	12
A	1	0	0	0
	1	0	0	0
	1	1	0	0
	0	1	1	0
	0	0	1	0
	0	0	0	1

1. L'euristica costruttiva seleziona, in ordine, le colonne 1,2,4 e 3.
2. La soluzione è ridondante: la colonna 2 può essere rimossa
3. L'euristica costruttiva ausiliaria rimuove la colonna 2 e fornisce una soluzione ottimale $x^* = \{1, 3, 4\}$

8.15.7 Estensione degli algoritmi costruttivi

Lo schema base degli algoritmi costruttivi può essere potenziato utilizzando:

1. un grafo di costruzione più efficace
 - aggiungere più di un elemento al corrente sottoinsieme x .
 - aggiungere elementi ad x , ma anche rimuovere elementi da x .
2. utilizzando un criterio di selezione più sofisticato:
 - funzioni **regret-based**, che stimano le potenziali future perdite associate con l'elemento i .
 - funzioni **look-ahead**, che stimano il valore finale dell'obiettivo ottenuto aggiungendo i ad x .

8.15.8 Euristiche costruttive regret-based

Le decisioni prese nei passi iniziali possono fortemente restringere le scelte fattibili nei passi successivi per via dei vincoli del problema.

- BPP: tutti gli oggetti devono essere messi in un container, ma gli assegnamenti iniziali possono rendere non disponibili alcuni container per oggetti successivi.

8 EURISTICHE COSTRUTTIVE

- TSP: tutti i nodi devono essere visitati, ma le decisioni di esplorazione iniziali possono rendere la visita dei nodi successivi più costosa.
- CMST: tutti i vertici devono essere collegati alla radice attraverso un sotto-albero, ma i collegamenti iniziali potrebbero rendere alcuni sotto-alberi non disponibili per i vertici successivi.

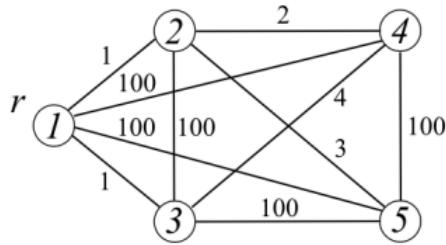
il criterio di selezione può tenerne conto implicitamente:

- BPP: l'euristica Decreasing First-Fit assegna gli oggetti più grande per prima.
- TSP: l'euristica Farthest Insertion visita i nodi più lontani per prima.

Alcuni criteri di selezione mirano esplicitamente a lasciare insiemi più ampi di buone scelte. Una tipica euristica regret-based consiste in:

- partizionare $\Delta_A^+(x)$ in classi disgiunte di scelte
- calcolare un criterio di scelta base per tutte le scelte.
- calcolare per ogni classe la differenza (*regret*) tra la scelta migliore e la seconda scelta migliore (oppure la media delle altre scelte, possibilmente pesata). Questo stima il danno subito rimandando la scelta migliore.
- scegliere la scelta migliore delle classi per la quale il "rimpianto" (criterio regret) è il massimo.

Per esempio, consideriamo il CMSTP ed il ground set $B = V \times T$, tale che ogni elemento sia composto dalla coppia $\langle \text{vertice}, \text{sottoalbero} \rangle$. Lasciamo che i pesi siano uniformi, quindi $w_v = 1 \forall v \in V$ e che la capacità sia $W = 2$.



Lasciamo che lo spazio di ricerca \mathcal{F} includa tutte le soluzioni parziali. L'algoritmo greedy mette il vertice 2 nel sotto-albero 1, il vertice 3 nel sotto-albero; il vertice 4 nel sotto-albero 1 e finalmente il vertice 5 nel sotto-albero 2:

$$c(x) = 1 + 1 + 2 + 100 = 104$$

L'algoritmo regret mette il vertice 2 nel sotto-albero 1, il vertice 3 nel sotto-albero 2; poi:

8 EURISTICHE COSTRUTTIVE

- il "rimpianto" del vertice 4 è la differenza $c(4, 2) - c(4, 1) = 4 - 2 = 2$
- il "rimpianto" del vertice 5 è la differenza $c(5, 2) - c(5, 1) = 100 - 3 = 97$

L'algoritmo mette il vertice 5 nel sotto-albero 1 ed in fine il vertice 4 nel sotto-albero 2:

$$c(x) = 1 + 3 + 1 + 4 = 9$$

8.15.9 Euristica costruttiva roll-out

Tale euristica costruttiva è anche conosciuta con il nome di **single-step look-ahead**, e fu proposta da Bertsekas e Tsitsiklis (1997).

Da un'euristica costruttiva base A :

- si inizia con un insieme vuoto $x^{(0)} = \emptyset$
- ad ogni passo t
 - si estende il sottoinsieme in una maniera fattibile: $x^{(t-1)} \cup \{i\}, \forall i \in \Delta_A^+(x)$
 - applicare l'euristica base ad ogni sottoinsieme esteso e calcolare la soluzione risultante $x_A(x^{(t-1)} \cup \{i\})$
 - utilizzare il valore della soluzione come criterio di selezione per scegliere $i^{(t)}$

$$\varphi_A(i, x) = f(x_A(x^{(t-1)} \cup \{i\}))$$

- terminare quando $\Delta_A^+(x)$ è vuoto.

prova ogni mossa possibile, controlla il risultato (look-ahead), torna indietro e scegli la mossa. Il risultato dell'euristica di roll-out domina quello dell'euristica base.

La complessità rimane polinomiale, ma è molto più grande: nel casto peggiore $T_{roA} = |B|^2 T_A$

Consideriamo il seguente esempio:

c	25 6 8 24 12																														
A	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table>	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	1	0	1	1	0	1	0	0	1	0	1	0	0	0	1
1	1	0	0	0																											
1	1	0	0	0																											
1	1	1	0	0																											
1	0	1	1	0																											
1	0	0	1	0																											
1	0	0	0	1																											

8 EURISTICHE COSTRUTTIVE

1. si inizia con il sottoinsieme vuoto: $x^{(0)} = \emptyset$
2. per ogni colonna i , applicare l'euristica costruttiva iniziando dal sottoinsieme $x^{(0)} \cup \{i\} = \{i\}$
 - per $i = 1$, otteniamo $x_A(\{1\}) = \{1\}$ di costo $f_A(\{1\}) = 25$
 - per $i = 2$, otteniamo $x_A(\{2\}) = \{2, 3, 5, 4\}$ di costo $f_A(\{2\}) = 50$
 - per $i = 3$, otteniamo $x_A(\{3\}) = \{3, 2, 5, 4\}$ di costo $f_A(\{3\}) = 50$
 - per $i = 4$, otteniamo $x_A(\{4\}) = \{4, 2, 5\}$ di costo $f_A(\{4\}) = 43$
 - per $i = 5$, otteniamo $x_A(\{5\}) = \{5, 2, 3, 4\}$ di costo $f_A(\{5\}) = 50$
3. la migliore soluzione è la prima, quindi $i^{(1)} = 1$
4. tutte le righe sono coperte: l'algoritmo termina.

Euristica roll-out generalizzata

Lo schema può essere generalizzato:

- applicare diverse euristiche base $A^{[1]}, \dots, A^{[l]}$
- incrementare il numero di passi look-ahead, ovvero usare $x^{(t-1)} \cup B^+$ con $|B^+| > 1$

Il risultato migliora e la complessità peggiora ulteriormente. Lo schema generale non cambia significativamente:

- si inizia dall'insieme vuoto: $x^0 = \emptyset$
- ad ogni passo t
 - per ogni possibile estensione $B^+ \in \Delta_A^+(x^{(t-1)})$ si applica ogni algoritmo base $A^{[l]}$ iniziando da $x^{(t-1)} \cup B^+$
 - il criterio di selezione è $\min_l f_{A^{[l]}}(x^{(t-1)} \cup B^+)$
 - si utilizza il valore della soluzione migliore come criterio di selezione per $i^{(t)}$
- quando $\Delta_A^+(x)$ è vuoto, termina.

8.16 Metaeuristiche costruttive

Gli algoritmi costruttivi hanno forti limitazioni su molti problemi, *cosa si può fare senza abbandonare lo schema generale?* Si può iterare lo schema generale per generare potenzialmente molte soluzioni diverse:

- **l'efficienza diminuisce:** i tempi computazionali vengono sommati.

8 EURISTICHE COSTRUTTIVE

- **l'efficacia aumenta:** la migliore soluzione viene restituita.

Quindi il *trade-off* deve essere attentamente impostato (in base al tempo computazionale e spaziale disponibile). Questo schema iterato si chiama **multi-start**, che come abbiamo detto applica differenti algoritmi ad ogni iterazione $I = 1, \dots, l$ (questo richiede di definire molteplici \mathcal{F}_{A_I} e φ_{A_I}). Il caso più semplice di metaeuristica che potrebbe anche non essere classificato come metaeuristica è quello precedentemente descritto ("meta" dal greco, "andare oltre"). Tuttavia, risulta più flessibile applicare metaeuristiche che sfruttano:

- **randomizzazione**, ovvero operazioni basate su un seme casuale (pseudo), come nel caso degli algoritmi semi-greedy, *GRASP* e *Ant System*.
- **memoria**, ovvero operazioni basate sulle soluzioni delle precedenti iterazioni, come nel caso del *ART*, *meccanismo di perturbazione dei costi* ed *Ant System*.

8.16.1 Condizione di terminazione

Prima di considerare l'approccio **multi-start**, parliamo della condizione di terminazione di quest'ultimo. Lo schema iterato può essere considerato idealmente infinito. In pratica si possono utilizzare condizioni di terminazione **absolute** o **relative**. Le prime:

1. un dato numero totale di esplorazioni del vicinato o un dato numero di ripetizioni della ricerca locale.
2. un dato tempo totale di esecuzione.
3. un dato valore dell'obiettivo.
4. un dato miglioramento dell'obiettivo rispetto alla soluzione di partenza o alla "relativa" condizione di terminazione.

Condizione di terminazione relative:

1. un dato numero di esplorazioni del vicinato o di ripetizioni dopo l'ultimo miglioramento di f^* .
2. un dato tempo di esecuzione dopo l'ultimo miglioramento
3. un dato valore minimo del rapporto tra miglioramento della funzione obiettivo e numero di esplorazioni o tempo d'esecuzione.

8.16.2 Multi-start

Il multi-start (o **restart**) è un approccio molto semplice e naturale:

- definisce differenti spazi di ricerca $\mathcal{F}_{A^{[I]}}$ ed il criterio di selezione $\varphi_{A^{[I]}}(i, x)$
- applico ogni algoritmo risultante A^l per ottenere la relativa soluzione $x^{[I]}$

8 EURISTICHE COSTRUTTIVE

- restituisco la soluzione migliore $x = \arg \min_{I=1,\dots,l} f(x^{[I]})$

un caso tipico è quello di regolare $\varphi_A(i, x)$ con parametri numeri μ . Il grafo di costruzione può modellare questa situazione con:

- includendo tutti i nodi e archi ammessi da almeno un algoritmo $A^{[I]}$

$$\mathcal{F}_A = \bigcup_{I=1}^l \mathcal{F}_A^{[I]}$$

- impostare il peso degli archi dipendendo su I : $\varphi_A(i, x, I) = \varphi_A^{[I]}(i, x)$
- impostare un peso infinito sugli archi che sono proibiti in un algoritmo specifico $A^{[I]}$: $\varphi_A(i, x, I) = +\infty$

Esempio

Un'intera famiglia di euristiche per il TSP può essere ottenuta impostando:

- il criterio di inserzione:

$$i_k^* = \arg \min_{i \in \{1, \dots, |x|\}} \gamma : i, k = \mu_1(c_{s_i, k} + c_{k, s_{i+1}}) - (1 - \mu_1)c_{s_i, s_{i+1}}$$

dove $\mu_1 \in [0; 1]$ regola la relativa forza:

- dell'incremento in costo dato dall'aggiunta del nodo k .
- del decremento in costo dato dalla rimozione dell'arco (s_i, s_{i+1})

- il criterio di selezione:

$$k^* = \arg \min_{k \in N \setminus N_x} \varphi_A(k, x) = \mu_2 d(x, k) - \mu_3 d(x, k) + (1 - \mu_2 - \mu_3) \gamma_{i_k^*, k}$$

dove $\mu_2, \mu_3 \in [0; 1]$ regola la relativa forze (ed il segno):

- della distanza del nodo k aggiunto rispetto al ciclo corrente x .
- dell'incremento in costo dato dall'aggiunta del nodo k .

per ottenere CI per $\mu = 1/2, 0, 0$, NI per $\mu = 1/2, 1, 0$, FI per $\mu = (1/2, 0, 1)$

8.16.3 Principali metaeuristiche costruttive

1. **Tecnica della ricerca adattiva** (*Adaptive Research Technique*, ART o Tabu Greedy): vieta alcune mosse in base alle soluzioni delle precedenti iterazioni

$$\min_{i: x \cup \{i\} \in \mathcal{F}^{[I]}} \varphi_A(i, x)$$

con $\mathcal{F}^{[I]} = \mathcal{F}^{[I]} \left(x_A^{[1]}, \dots, x_A^{[I-1]} \right) \subseteq \mathcal{F}$
(molto meno popolare degli altri due).

8 EURISTICHE COSTRUTTIVE

2. **semigreedy** e **GRASP**: utilizzano un criterio di selezione randomizzato

$$\min_{i:x \cup \{i\} \in \mathcal{F}^{[I]}} \varphi_A(i, x, \omega^{[I]})$$

3. **Ant System** (AS): utilizzando un criterio di selezione randomizzato che dipende dalle soluzioni delle precedenti iterazioni

$$\min_{i:x \cup \{i\} \in \mathcal{F}^{[I]}} \varphi_A(i, x, \omega^{[I]}, x_A^{[1]}, \dots, x_A^{I-1})$$

le nuove informazioni su gli archi del grafo di costruzione guidano la ricerca. L'ART utilizza la memoria, il GRASP utilizza la randomizzazione, l'AS utilizza entrambi.

8.16.4 Adaptive Research Technique

L'ART fu proposto da Patterson nel 1998 per il CMSTP. Quando nei primi passi sono introdotti elementi ingannevolmente buoni (falsi positivi), la soluzione finale può essere molto brutta. Per evitare questo:

- l'approccio roll-out effettua un look-ahead su ogni possibile elemento (ma un singolo passo può essere insufficiente per individuare diversi elementi ingannevoli).
- l'ART nega alcuni elementi di guidare il sottoinsieme x sul giusto percorso nello spazio di ricerca.

Come identificare gli elementi ingannevoli? L'obiettivo è la diversificazione: vietare elementi delle soluzioni precedenti garantisce di ottenere soluzioni diverse. Le proibizioni sono temporanee, con un tempo di scadenza di L iterazioni; altrimenti, costruire soluzioni fattibili diventerebbe impossibile.

Algoritmo

Sia definita una euristica costruttiva base A . Sia T_i l'iterazione di proibizione per ogni elemento $i \in B$ e x^* sia la soluzione migliore. Impostare $T_i = -\infty \forall i \in B$ per indicare che nessuno elemento è vietato. Ad ogni iterazione $I \in \{1, \dots, l\}$

1. applicare l'euristica A vietando tutti gli elementi i tali che $I \leq T_i + L$ (tutte le proibizioni più vecchie di L scaderanno automaticamente); lasciamo che $x^{[I]}$ sia la soluzione risultante.
2. se $x^{[I]}$ è migliore di x^* , impostiamo $x^* := x^{[I]}$ e salva $T_i - I \forall i \in B$
3. decidere quale elemento vietare ed impostare $T_i = I$ per loro: ogni elemento è vietato con una probabilità π
4. fare minori aggiustamenti ad L, π o T_i .

al termine, restituire x^* .

8 EURISTICHE COSTRUTTIVE

Esempio, ART per SCP

c	25	6	8	24	12
A	1	1	0	0	0
	1	1	0	0	0
	1	1	1	0	0
	1	0	1	1	0
	1	0	0	1	0
	1	0	0	0	1

- l'euristica base trova la soluzione $x^{[1]} = \{2, 3, 5, 4\}$ di costo $f(x^{[1]}) = 50$; vietiamo (casualmente) la colonna 2.
- l'euristica base trova la soluzione $x^{[2]} = \{3, 1\}$ di costo $f(x^{[2]}) = 33$; vietiamo (casualmente) la colonna 3.
- l'euristica base trova la soluzione $x^{[3]} = \{1\}$ di costo $f(x^{[3]}) = 25$, la quale è una soluzione ottima.

Nel caso di un'esecuzione sfortunata al secondo step si potrebbe vietare la colonna 1.

Intensificazione

L'ART ha tre parametri base:

- il numero totale di iterazioni ℓ .
- la lunghezza L della proibizione.
- la probabilità π della proibizione.

Un'eccessiva diversificazione può nascondere la scoperta dell'ottimo, l'**intensificazione** mira a concentrare la ricerca sui sottoinsiemi più promettenti. La diversificazione e intensificazione giocano ruoli complementari. L'intensificazione può essere ottenuta regolando i parametri in base:

- i dati del problema: assegnando a gli elementi più promettenti (quelli che costano meno)
 - una probabilità π più piccola di essere evitato.
 - un tempo di scadenza L_i più corto della proibizione.
- memoria: per gli elementi promettenti (quelli che appaiono nelle soluzioni meglio conosciute)
 - riducendo L_i (se $L_i = 0$, i non è mai vietato).

8 EURISTICHE COSTRUTTIVE

- periodicamente riavviare l'algoritmo con i valori $T_i - I$ associati con la migliore soluzione nota, anziché $T_i = -\infty$.

Come assegnare dei valori efficaci ai parametri? Il confronto sperimentale di valori diversi è necessario ma complesso.

1. richiede lunghe campagne di sperimentazione, perché il numero di configurazioni cresce combinatoriamente con il numero di parametri ed il numero di valori testati per ogni parametro.
2. si rischia il *sovradattamento (overfitting)*, che consiste nell'etichettare come valori assolutamente "buoni" dei valori che sono buoni effettivamente solo su istanze del benchmark.

Questo eccesso di parametri è un aspetto indesiderabile, e spesso si rivela uno studio insufficiente del problema e dell'algoritmo.

8.16.5 Semi-greedy heuristics

Un algoritmo costruttivo non esatto ha almeno un passo t che costruisce un sottoinsieme $x^{(t)}$ non incluso in alcuna soluzione ottimale. Visto che l'elemento selezionato è il migliore in accordo con il criterio di selezione

$$i^* = \arg \min_{i \in \Delta_A^+(x)} \varphi_A(i, x)$$

necessariamente $\varphi_A(i, x)$ è incorretto, ma probabilmente non completamente sbagliato. L'algoritmo semi-greedy (Hart e Shogan, 1987) assume che gli elementi che portano all'ottimo sono molto buoni per $\varphi_A(i, x)$, anche se non strettamente i migliori.

Se non è possibile ridefinire $\varphi_A(i, x)$:

- definire una probabilità di distribuzione adatta su $\Delta_A^+(x)$ che favorisce gli elementi con i migliori valori di $\varphi_A(i, x)$
- selezionare $i^*(\omega)$ in accordo alla funzione di distribuzione.

Visto che l'insieme di possibili alternative è finito, questo significa assegnare:

- una probabilità $\pi_A(i, x)$ all'arco $(x, x \cup \{i\})$ del grafo di costruzione (con una somma uguale ad 1 per gli archi uscenti da ogni nodo)

$$\sum_{i \in \Delta_A^+(x)} \pi_A(i, x) = 1 \quad \forall x \in \mathcal{F}_A : \Delta_A^+(x) \neq \emptyset$$

- alte probabilità agli elementi migliori per il criterio di selezione

$$\varphi_A(i, x) \leq \varphi_A(j, x) \Leftrightarrow \pi_A(i, x) \geq \pi_A(j, x)$$

per ogni $i, j \in \Delta_A^+(x), x \in \mathcal{F}_A$

8 EURISTICHE COSTRUTTIVE

Questo approccio euristico ha delle proprietà importanti:

- può raggiungere una soluzione ottima se è presente un cammino da \emptyset a X^*
- può essere riapplicato diverse volte ottenendo delle soluzioni differenti e la probabilità di raggiungere la soluzione ottima cresce gradualmente.

Convergenza verso l'ottimo

La probabilità di:

- seguire il cammino γ è il prodotto delle probabilità su gli archi

$$\prod_{(y,y \cup \{i\} \in \gamma)} \pi_A(i,y)$$

- ottenere una soluzione x corrisponde alla sommatoria dei cammini Γ_x che raggiungono x

$$\sum_{\gamma \in \Gamma_x} \prod_{(y,y \cup \{i\} \in \gamma)} \pi_A(i,y)$$

Questo implica che la probabilità di raggiungere l'ottimo:

1. non è zero se e solo se esiste un percorso con una probabilità non zero da \emptyset a X^* .
2. incrementa per $\ell \rightarrow +\infty$ (la probabilità di non raggiungerlo decremente gradualmente).

In questo contesto, una *camminata casuale* è una metaeuristiche costruttiva nella quale tutti gli archi uscenti rispetto allo stesso nodo hanno uguale probabilità.

- si trova un cammino verso l'ottimo con probabilità 1 (se esiste).
- il tempo richiesto può essere estremamente lungo (l'algoritmo esaustivo è esatto e richiede tempo finito).

Un'euristica costruttiva deterministica imposta tutte le probabilità a 0, eccetto per quelle su gli archi di un singolo cammino: troverà la soluzione ottima solo se verranno soddisfatte delle proprietà specifiche.

Le euristiche randomizzate che favoriscono gli archi promettenti e penalizzano gli altri:

- accelerano il tempo di convergenza medio.
- decrementano la garanzia di convergenza nel caso peggiore.

C'è un trade-off tra risultato desiderati e peggiori. Gli archi con zero probabilità possono bloccare il cammino verso l'ottimo. Gli archi con probabilità di convergere verso zero riducono la probabilità di trovarlo.

8.16.6 Semi-greedy e GRASP

GRASP, sta per *Greedy Randomized Adaptive Search Procedure*, è una sofisticata variante di euristiche semi-greedy.

- Greedy: perché utilizza un'euristica costruttiva base.
- Randomized: perché l'euristica base effettua dei passi casuali.
- Adaptive: perché l'euristica utilizza un criterio di selezione adattivo $\varphi_A(i, x)$, che dipende anche da x (non per forza).
- Search: perché alterna l'euristica costruttiva e l'euristica di scambio.

L'utilizzo dell'euristica ausiliare di scambio permette di ottenere risultati migliori.

Funzione probabilistica

Diverse funzioni $\pi_A(i, x)$ sono monotone rispetto a $\varphi_A(i, x)$

$$\varphi_A(i, x) \leq \varphi_A(j, x) \leftarrow \pi_A(i, x) \geq \pi_A(j, x)$$

- **Probabilità uniforme:** ogni arco uscente da x ha la stessa probabilità $\pi_A(i, x)$. L'algoritmo esegue un cammino casuale in \mathcal{F}_A (*random walk*).
- **HBSS (Heuristic-Biased Stochastic Sampling):** si ordinano gli archi uscenti di x per valori non-decrescenti di $\varphi_A(i, x)$. Si assegna una probabilità decrescente in base alla posizione in base ad uno schema semplice (lineare, esponenziale, ...).
- **RCL (Restricted Candidate List):** si ordinano tutti gli archi uscenti di x per valori non decrescenti di $\varphi_A(i, x)$. Si inseriscono gli archi migliori in una lista, e si assegna una probabilità uniforme a gli archi della lista, mentre una probabilità zero a gli altri.

La strategia più comune è la RCL, anche se gli archi con probabilità zero potrebbero potenzialmente cancellare la convergenza globale all'ottimo.

8 EURISTICHE COSTRUTTIVE

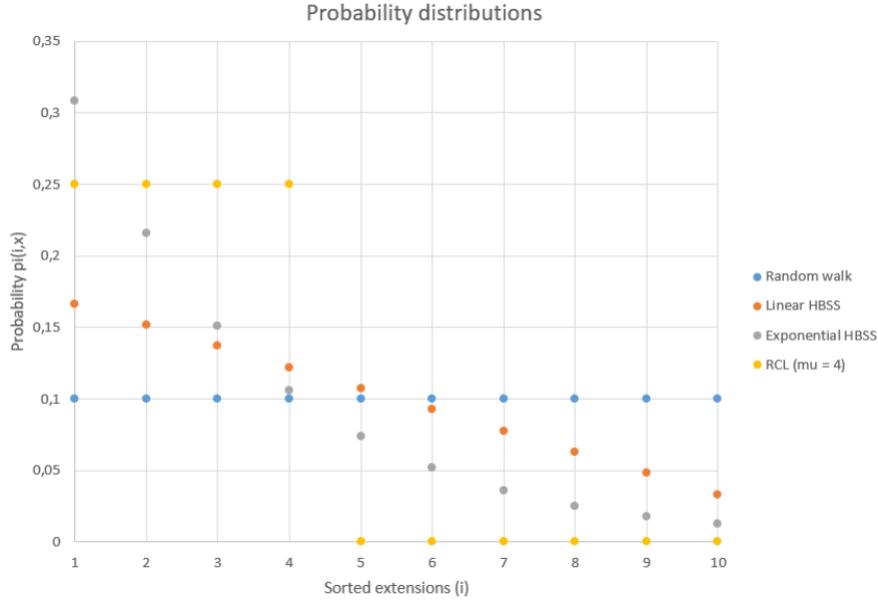


Figura 8.29: Ipotizziamo che al passo corrente possano essere aggiunti 10 elementi, $|\Delta_A^+(x)| = 10$

Sono presenti due strategie principali per definire l'RCL:

- **cardinalità:** l'RCL include i migliori μ elementi di $\Delta_A^+(x)$, dove $\mu \in \{1, \dots, |\Delta_A^+(x)|\}$ è un parametro fissato dall'utente.
 - $\mu = 1$ restituisce l'euristica costruttiva base.
 - $\mu = |B|$ restituisce una *random walk*.
- **valore:** l'RCL include tutti gli elementi di $\Delta_A^+(x)$ il cui valore si trova tra φ_{min} e $(1 - \mu)\varphi_{min} + \mu\varphi_{max}$ dove:

$$\varphi_{min}(x) = \min_{i \in \Delta_A^+(x)} \varphi_A(i, x) \quad \varphi_{max}(x) = \varphi_A(i, x)$$

e $\mu \in [0; 1]$ è un parametro fissato dall'utente:

- $\mu = 0$ restituisce un'euristica costruttiva base.
- $\mu = 1$ restituisce una *random walk*.

8 EURISTICHE COSTRUTTIVE

Algorithm 8 GRASP(I)

```

 $x := B$ 
 $x* := +\infty$ 
for  $I = 1$  to  $\ell$  do
    Constructive heuristic with random steps
     $x := \emptyset;$ 
    while  $\Delta_A^+(x) \neq \emptyset$  do
         $\varphi_i := \varphi_A(i, x)$  for each  $i \in \Delta_A^+(x)$ 
         $\pi := \text{AssignProbabilities}(\Delta_A^+(x), \varphi, \mu);$ 
         $i := \text{RandomExtract}(\Delta_A^+(x), \pi);$ 
         $x := x \cup \{i\};$ 
    end while
    if  $x \in X$  and  $f(x) < f^*$  then
         $x^* := x;$ 
         $f^* := f(x);$ 
    end if
end for
Return( $x^*, f^*$ );

```

Esempio GRASP per SCP

c	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>25</td><td>6</td><td>8</td><td>24</td><td>12</td></tr> </table>	25	6	8	24	12																									
25	6	8	24	12																											
A	<table border="1" style="border-collapse: collapse; width: 100%; text-align: center;"> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table>	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	1	0	1	1	0	1	0	0	1	0	1	0	0	0	1
1	1	0	0	0																											
1	1	0	0	0																											
1	1	1	0	0																											
1	0	1	1	0																											
1	0	0	1	0																											
1	0	0	0	1																											

1. si inizia con un insieme vuoto $x^{(0)} = \emptyset$
2. si costruisce un RCL con $\mu = 2$ candidati: ovvero le colonne 2 ($\varphi_2 = 2$) e 3 ($\varphi_3 = 4$); si seleziona casualmente la 3.
3. si costruisce un RCL con $\mu = 2$ candidati: ovvero le colonne 2 ($\varphi_2 = 3$) e 1 ($\varphi_3 = 6.25$); si seleziona casualmente la 1.
4. la soluzione ottenuta è $x = \{3, 1\}$ di costo $f(x) = 33$

Con $\mu = 2$, la soluzione ottima non può essere ottenuta; con $\mu = 3$ può essere ottenuta, è possibile anche con $\mu = 2$ se una fase distruttiva viene applicata.

8 EURISTICHE COSTRUTTIVE

8.16.7 Algoritmo semi-greedy reattivo

Ancora una volta sono presenti dei parametri da regolare:

- il numero di iterazioni ℓ
- il valore μ che determina la dimensione del RCL

Un'idea di sfruttare la memoria è quella di *imparare* dai risultati precedenti:

1. selezionare m configurazioni dei parametri μ_1, \dots, μ_m e l'insieme $\ell_r = \ell/m$
2. si esegue ogni configurazione μ_r per ℓ_r iterazioni.
3. si calcola il significato di $\bar{f}(\mu_r)$ dei risultati ottenuti con μ_r
4. si aggiorna il numero di iterazioni ℓ_r per ogni μ_r basato su $\bar{f}(\mu_r)$

$$\ell_r = \frac{\frac{1}{\bar{f}(\mu_r)}}{\sum_{s=1}^m \frac{1}{\bar{f}(\mu_s)}} \ell \quad \text{per } r = 1, \dots, m$$

incrementandole per le configurazioni più efficaci.

5. si ripete l'intero processo, si torna al punto 2, per R volte. Altri schemi utilizzano punti basati sul numero dei risultati migliori conosciuti.

8.16.8 Metodi di perturbazione dei costi

Anziché evitare/forzare delle scelte, o modificarne la loro probabilità, è possibile modificare l'attrazione di quelle disponibili. Dato un'economia costruttiva A , ad ogni passo dell'iterazione I

- si regola il criterio di selezione $\varphi_A(i, x)$ con un fattore $\tau_A^{[I]}(i, x)$

$$\Psi_A^{[I]}(i, x) = \frac{\varphi_A(i, x)}{\tau_A^{[I]}(i, x)}$$

- si aggiorna $\tau_A^{[I]}(i, x)$ in base alle soluzioni precedenti $x^1, \dots, x^{[I-1]}$

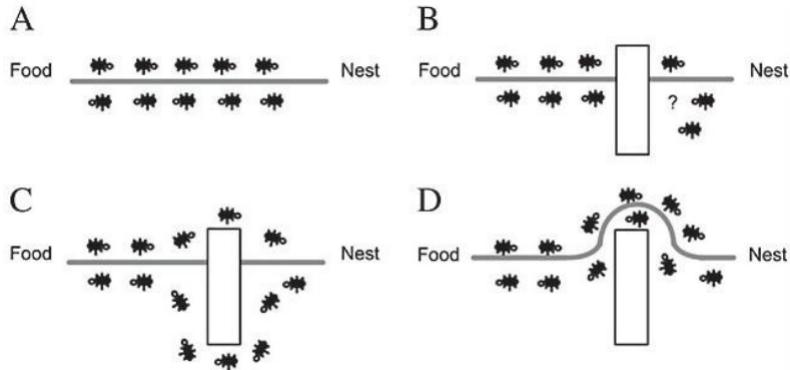
Gli elementi con una migliore $\varphi_A(i, x)$ tendono ad essere favoriti, ma $\tau_A^{[I]}(i, x)$ regola questo effetto, promuovendo:

- **intensificazione** se $\tau_A^{[I]}(i, x)$ incrementa per gli elementi più frequenti; questo favorisce soluzioni *simili* a quelle precedenti.
- **diversificazione** se $\tau_A^{[I]}(i, x)$ decremente per gli elementi più frequenti; questo favorisce soluzioni *differenti* dalle precedenti.

8 EURISTICHE COSTRUTTIVE

8.16.9 Ottimizzazione della colonia di formiche

Questo approccio è stato ideato da Dorigo, Maniezzo e Colomi nel 1991, prendendo ispirazione dal comportamento sociale delle formiche. La **stigmergia** è la comunicazione indiretta attraverso diversi agenti che sono influenzati dai risultati delle azioni (come modifiche dell'ambiente) di tutti gli agenti.



Ogni agente è un applicazione dell'euristica costruttiva base:

- lascia un sentiero sui dati che dipende dalla soluzione generata.
- esegue delle scelte influenzate dai sentieri lasciati da gli altri agenti.

Le scelte dell'agente hanno una componente casuale.

Sentiero

Come nell'euristica semi-greedy:

- viene fornita un'euristica costruttiva di base A
- ogni passo esegue una scelta casualmente parziale

In maniera differente dalle euristiche semi-greedy:

- ogni iterazione I esegue h volte l'euristica A (popolazione).
- tutte le scelte di $\Delta_A^+(x)$ sono fattibili (non esiste RCL).
- La probabilità $\pi_A(i, x)$ dipende da:
 - criterio di selezione $\varphi_A(i, x)$
 - informazioni ausiliarie $\tau_A(i, x)$ denotate come un sentiero prodotto dalle iterazioni precedenti (a volte da altri agenti nella stessa iterazione).

Il sentiero è uniforme all'inizio ($\tau_A(i, x) = \tau_0$), e dopo regolato:

8 EURISTICHE COSTRUTTIVE

- incrementando in favore di scelte promettenti.
- decrementando per evitare scelte ripetitive.

Per semplicità, il sentiero $\tau_A(i, x)$ non è associato ad ogni arco $(x, x \cup \{i\})$, ma è lo stesso per i blocchi degli archi.

Scelte casuali

Anziché selezionare il migliore elemento accordandosi con il criterio $\varphi_A(i, x)$, i è estratto da $\Delta_A^+(x)$ con probabilità:

$$\pi_A(i, x) = \frac{\tau_A(i, x)^{\mu_\tau} \eta_A(i, x)^{\mu_\eta}}{\sum_{j \in \Delta_A^+(x)} \tau_A(j, x)^{\mu_\tau} \eta_A(j, x)^{\mu_\eta}}$$

dove:

- il denominatore normalizza la probabilità.
- la **visibilità** è rappresentata dalla funzione ausiliaria:

$$\eta_A(i, x) = \begin{cases} \varphi_A(i, x) & \text{per i problemi di massimizzazione} \\ \frac{1}{\varphi_A(i, x)} & \text{per i problemi di minimizzazione} \end{cases}$$

- i parametri μ_τ e μ_η regolano i pesi dei due termini

I parametri μ_η e μ_τ regolano il peso dei dati e la memoria:

- $\mu_\eta \approx 0$ e $\mu_\tau \approx 0$ spingono verso la casualità.
- $\mu_\eta \gg \mu_\tau$ preferisce i dati, il che ha senso quando le soluzioni conosciute non sono molto significative.
- $\mu_\eta \ll \mu_\tau$ preferisce la memoria, quindi chiude rispetto le soluzioni precedenti, il che ha senso quando le soluzioni conosciute sono molto significative.

Il sistema **ANT** (*Ant Colony System*) semplifica lo schema impostando $\mu_\eta = \mu_\tau = 1$

$$\pi_A(i, x) = \frac{\tau_A(i, x) \eta_A(i, x)}{\sum_{j \in \Delta_A^+(x)} \tau_A(j, x) \eta_A(j, x)}$$

e selezionando il nuovo elemento $i^{(t)}$ al passo t :

- casualmente con una probabilità q
- ottimizzando $\varphi_A(i, x)$ con probabilità $(1 - q)$

così che:

- $q \approx 0$ favorisce i dati.
- $q \approx 1$ favorisce la memoria.

8 EURISTICHE COSTRUTTIVE

Aggiornamento del sentiero

Ad ogni iterazione ℓ :

1. si eseguono h istanze dell'euristica base A .
2. si selezionano un sotto insieme $\tilde{X}^{[I]}$ di soluzioni ottenute, in ordine di favori gli elementi nelle seguenti iterazioni.
3. aggiornare il sentiero utilizzando la formula:

$$\tau_A(i, x) := (1 - \rho)\tau_A(i, x) + \rho \sum_{y \in \tilde{X}^{[I]}: i \in y} F_A(y)$$

dove:

- $\rho \in [0; 1]$ è un **parametro di oblio**.
- $F_A(y)$ è una funzione **fitness** che esprime la qualità della soluzione y .

Lo scopo dell'aggiornamento è di:

1. incrementare il sentiero su gli elementi delle specifiche soluzioni ($y \in \tilde{X}^{[I]}$).
2. decrementare il sentiero su gli altri elementi.

Il parametro di oblio

Il parametro di oblio $\rho \in [0; 1]$ regola il comportamento dell'algoritmo:

- **diversificazione:** un **oblio alto** ($\rho \approx 1$) cancella il sentiero corrente basato sull'intuizione che: le soluzioni ottenute non sono affidabili, e che differenti soluzioni andrebbero esplorate.
- **intensificazione:** un **oblio basso** ($\rho \approx 0$) preserva il sentiero corrente basando sull'intuizione che le soluzioni ottenute siano affidabili, e che le soluzioni simili dovrebbero essere esplorate.

$\tilde{X}^{[I]}$ colleziona le soluzioni attorno alle quali la ricerca verrà intensificata:

- il classico Ant System considera tutte le soluzioni di iterazione $I - 1$.
- il metodo elitario considera le soluzioni meglio conosciute:
 - le soluzioni migliori dell'iterazione $I - 1$
 - le soluzioni migliori di tutte le iterazioni $< I$

I metodi elitari:

- trovano migliori risultati in meno tempo.
- richiedono meccanismi addizionali per evitare convergenze premature.

8 EURISTICHE COSTRUTTIVE

8.16.10 Varianti dell'Ant System

- $\mathcal{MAX} - \mathcal{MIN}$ Ant System: impone che sul sentiero un range di valori $[\tau_{min}, \tau_{max}]$, sperimentalmente regolati.
- HyperCube Ant Colony Optimization (HC-ACO): normalizza il sentiero tra 0 e 1.
- **Ant Colony System:** aggiorna il sentiero su due livelli
 - **Aggiornamento globale** (già visto) modifica ad ogni iterazione \uparrow . Lo scopo è quello di intensificare la ricerca.
 - **Aggiornamento locale** aggiorna il sentiero ad ogni applicazione di } dell'euristica base in modo da scoraggiare le scelte identiche nella seguente maniera

$$\tau_A(i, x) := (1 - \rho)\tau_A(i, x) \quad \forall i \in x_A^{[I,g]}$$

Algorithm 9 Algoritmo AntSystem

```

 $x^* := \emptyset$ 
 $f^* := +\infty$ 
for  $I = 1$  to  $\ell$  do
  for  $g = 1$  to  $h$  do
     $x := A(I, \tau_A)$ 
     $x := Search(x)$ 
    if  $f(x) < f^*$  then
       $x^* := x$ 
       $f^* := f(x)$ 
       $\tau_A := LocalUpdate(\tau_A, x)$ 
    end if
  end for
   $\tilde{X}^{[I]} := Update(\tilde{X}^{[I]}, x)$ 
   $\tau_A := GlobalUpdate(\tau_A, \tilde{X}^{[I]})$ 
end for
Return  $(x^*, f^*)$ 

```

Convergenza verso la soluzione ottima

Alcune variante dell'Ant System convergono verso la soluzione ottima con una probabilità 1 (Gutjahr, 2002). L'analisi è basata sul grafo di costruzione:

- il sentiero $\tau_A(i, x)$ si adagia su gli archi $(x, x \cup \{i\})$
- nessuna informazione dai dati viene utilizzata, che $\eta_A(i, x) \equiv 1$ (questa strana assunzione semplifica la computazione, ma non è necessaria).
- $\tau^{[I]}$ è la funzione sentiero all'inizio dell'iterazione I .

8 EURISTICHE COSTRUTTIVE

- $\gamma^{[I]}$ è il miglior percorso sul grafo al termine dell'iterazione I .
- $(\tau^{[I]}, \gamma^{[I-1]})$ è lo stato di un processo di Markov non omogeneo: la probabilità di ogni stato dipende solo dall'iterazione precedente, il processo non è omogeneo perché la dipendenza varia con I .

La dimostrazione conclude che per $\ell \rightarrow +\infty$ con probabilità 1

1. Almeno una corsa segue il percorso ottimo all'interno dello spazio di ricerca.
2. Il sentiero τ tende ad un massimo lungo uno dei percorsi ottimali, e a zero sugli altri archi.

Prima variante con convergenza globale Il sentiero è aggiornato con un coefficiente di oblio variabile

$$\tau^{[I]}(I, x) := \begin{cases} (1 - \rho^{[I-1]})\tau^{[I-1]}(i, x) + \rho^{[I-1]} \frac{1}{|\gamma^{[I-1]}|} & \text{if } (x, x \cup \{i\}) \in \gamma^{[I-1]} \\ (1 - \rho^{[I-1]})\tau^{[I-1]}(i, x) & \text{otherwise} \end{cases}$$

dove $\gamma^{[I-1]}$ è il miglior percorso trovato nel grafo all'iterazione $I - 1$ con $|\gamma^{[I-1]}|$ come numero dei suoi archi. Se la variabile di oblio decrementa abbastanza

$$\rho^{[I]} \leq 1 - \frac{\log I}{\log(I+1)} \text{ and } \sum_{I=0}^{+\infty} \rho^I = +\infty$$

con una probabilità 1 lo stato converge in (τ^*, γ^*) , dove

- γ^* è un percorso ottimale nel grafo di costruzione.
- $\tau^*(i, x) = \frac{1}{|\gamma^*|}$ per $(x, x \cup \{i\}) \in \gamma^*$, altrimenti 0

Seconda variante con convergenza globale Alternativamente, nel caso in cui l'oblio ρ rimanga costante, ma il sentiero viene forzato da una soglia minima che decresce lentamente

$$\tau(i, x) \geq \frac{c_I}{\log(I+1)} \text{ and } \lim_{I \rightarrow +\infty} c_I \in (0; 1)$$

con una probabilità 1 lo stato converge in (τ^*, γ^*) , l'oblio è ristretto ad una soglia minima.

In pratica, tutti gli algoritmi proposti fin'ora:

- associare il sentiero ad un gruppo di archi $(x, x \cup \{i\})$
- utilizzare valori costanti per i parametri ρ e τ_{min}

non garantiscono la convergenza, sia τ che π possono tendere a zero su ogni percorso ottimale.

9 Euristiche di scambio

9.1 Algoritmi di scambio

In ottimizzazione combinatoria ogni soluzione x è un sottoinsieme di B . Un'euristica di scambio aggiorna il sottoinsieme corrente $x^{(t)}$ passo per passo.

1. si inizia da una soluzione fattibile $x^{(0)} \in X$ trovata in qualche modo (solitamente con un'euristica costruttiva).
2. si genera una **famiglia di soluzioni fattibili** scambiando gli elementi, in sostanza si aggiungono i sottoinsiemi A esterni a $x^{(t)}$ e si eliminano i sottoinsiemi D interni a $x^{(t)}$.

$$x'_{A,D} = x \cup A \setminus D$$

con $A \subseteq B \setminus x$ e $D \subseteq x$

3. si utilizza un criterio di selezione $\varphi(x, A, D)$ per scegliere i sottoinsiemi da scambiare

$$(A^*, D^*) = \text{ARG}_{(A,D)} \min \varphi(x, A, D)$$

4. eseguire la scelta selezionata per generare una nuova soluzione corrente

$$x^{(t+1)} := x^{(t)} \cup A^* \setminus D^*$$

5. se una condizione di terminazione viene raggiunta, termina; altrimenti, ritorna al punto 2.

Un'euristica di scambio è definita da:

- la coppia di sottoinsiemi scambiabili (A, D) in tutte le soluzioni x , sostanzialmente le soluzioni generate da un singolo scambio a partire da x .
- il criterio di selezione $\varphi(x, A, D)$.

La funzione vicinato (in inglese "*neighbourhood*") $N : X \rightarrow 2^X$ è una funzione che associa ad ogni possibile soluzione $x \in X$ un sottoinsieme di soluzioni fattibili $N(x) \subseteq X$. La situazione può essere formalmente descritta con un grafo di ricerca nel quale:

- i nodi rappresentano le soluzioni fattibili $x \in X$
- gli archi connettono ogni soluzione x a quelle del suo vicino $N(x)$, muovendo gli elementi dentro e fuori x (ci si riferisce con "mosse").

Ogni esecuzione dell'algoritmo corrisponde ad un cammino nel grafo di ricerca. *Come si definisce un algoritmo che corrisponde ad un cammino in un grafo di ricerca?*

9 EURISTICHE DI SCAMBIO

9.1.1 Vicinato basato sulla distanza

Ogni soluzione $x \in X$ può essere rappresentata da il suo **vettore di incidenza** (incidence vector).

$$\xi_i(x) = \begin{cases} 1 & \text{if } i \in x \\ 0 & \text{if } i \in B \setminus x \end{cases}$$

La distanza di Hamming tra due soluzioni x e x' è il numero di elementi nel quale i vettori di incidenza differiscono

$$d_H(x, x') = \sum_{i \in B} |\xi_i(x) - \xi_i(x')|$$

Riferendosi ai sottoinsiemi, $d_H(x, x') = |x \setminus x'| + |x' \setminus x|$ Una tipica definizione di vicinato, con un parametro intero k , è l'insieme di tutte le soluzioni con una distanza di Hamming da x non più grande di k .

$$N_{H_k}(x) = \{x' \in X : d_H(x, x') \leq k\}$$

Esempio con il KP

Il KP è composto dalla seguente istanza $B = \{1, 2, 3, 4\}$, $v = [5 4 3 2]$ e $V = 10$, ha 13 soluzioni possibili su 16 sottoinsiemi.

(0111)	(1111)	(0001)	(0000)
(0011)	(1011)	(1010)	(0110)
(1110)	(1001)	(1000)	(0101)
(1100)	(1000)	(0010)	(0100)

visto che i sottoinsiemi $\{1, 2, 3, 4\}$, $\{1, 2, 3\}$ e $\{1, 2, 4\}$ sono infattibili. La soluzione $\{1, 3, 4\}$ (denotata in blu) ha un vicinato $N_{H_2}(x)$ che consiste di 7 elementi (in rosa). Il sottoinsieme in nero non appartiene al vicinato, perché la distanza di Hamming da x è > 2 .

9.1.2 Vicinato basato sulle operazioni

Un'altra definizione comune di vicinato è ottenuta definendo:

- una famiglia \mathcal{O} di operazioni sulle soluzioni del problema.
- l'insieme di tutte le soluzioni generate applicando ad x le operazioni \mathcal{O}

$$N_{\mathcal{O}} = \{x' \in X : \exists o \in \mathcal{O} : o(x) = x'\}$$

Considerando ancora il KP, \mathcal{O} può essere definita come:

9 EURISTICHE DI SCAMBIO

- aggiungere ad x un elemento di $B \setminus x$
- rimuovere da x un elemento.
- scambiare un elemento di x con uno di $B \setminus x$

Il vicinato risultante $N_{\mathcal{O}}$ è relativo a quello definito dalla distanza di Hamming, ma non coincide con nessuna di queste

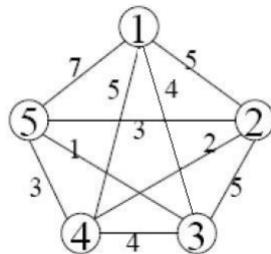
$$N_{H_1} \subset N_{\mathcal{O}} \subset N_{H_2}$$

Come nel vicinato basato sulle distanze, i vicini possono essere parametrizzati considerando sequenze di k operazioni di \mathcal{O} anziché una singola

$$N_{\mathcal{O}_k}(x) = \{x' \in X : \exists o_1, \dots, o_k \in \mathcal{O} : o_k(o_{k-1}(\dots o_1(x))) = x'\}$$

9.1.3 Vicinato basato sulla distanza e sulle operazioni

In generale, un vicinato basato sulle operazioni include soluzioni con differenti distanze di Hamming da x . Per il TSP uno può definire un vicinato N_{S_1} includendo le soluzioni ottenute scambiando due nodi nell'ordine di visita.



Il vicinato di soluzione $x = (3, 1, 4, 5, 2)$ è:

$$N_{S_1}(x) = \{(\textcolor{red}{1}, 3, 4, 5, 2), (\textcolor{red}{4}, 1, 3, 5, 2), (\textcolor{red}{5}, 1, 4, 3, 2), (\textcolor{red}{2}, 1, 4, 5, 3), (\textcolor{red}{3}, 4, 1, 5, 2), (\textcolor{red}{3}, 5, 4, 1, 2), (\textcolor{red}{3}, 2, 4, 5, 1), (\textcolor{red}{3}, 1, 5, 4, 2), (\textcolor{red}{3}, 1, 2, 5, 4), (\textcolor{red}{3}, 1, 4, 2, 5)\}$$

Figura 9.1: s
e due nodi sono adiacenti, gli archi modificati sono 3+3; altrimenti, loro sono 4+4.

a volte le due definizioni comportano lo stesso vicinato:

- per il MDP
 - il vicinato N_{H_2} (soluzioni con una distanza di Hamming uguale a 2)
 - il vicinato N_{S_1} (scambia un elemento di x con uno di $B \setminus x$)
- per il BPP

9 EURISTICHE DI SCAMBIO

- il vicinato N_{H_2}
- il vicinato N_{T_1} (trasferisce un oggetto in un container differente)
- per il Max-SAT
 - il vicinato N_{H_2}
 - il vicinato N_{F_1} ("flip" di una variabile: inverte l'assegnamento di verità)

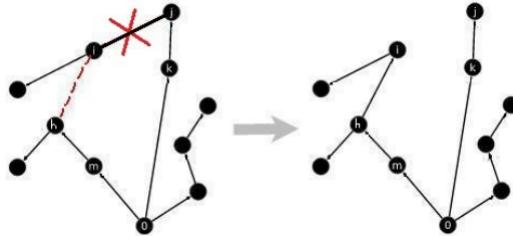
Questo è tipico dei problemi con soluzioni a cardinalità fissa:

- effettuano una sequenza di k scambi tra i singoli elementi ($|A| = 1 = |D| = 1$): k elementi vanno in x e k elementi secondo da x .
- La distanza di Hamming tra due soluzioni estreme è $\leq 2k$

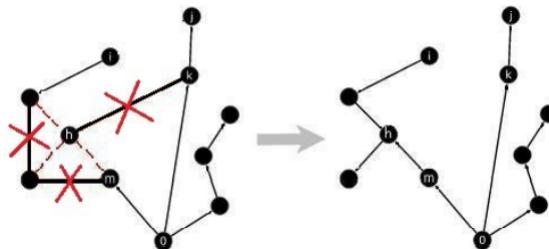
Vicinato differente per lo stesso problema: CMST

Differenti ground set restituiscono differenti vicinati. Nel CMST è possibile impostare $B = E$ o $B = V \times T$.

- Scambiando gli archi: rimuovo (i, j) , aggiungendo (i, h)



- Scambiando i vertici: sposto il nodo h dal ramo 2 al ramo 1, e ricalcolo il minimo albero ricoprente.

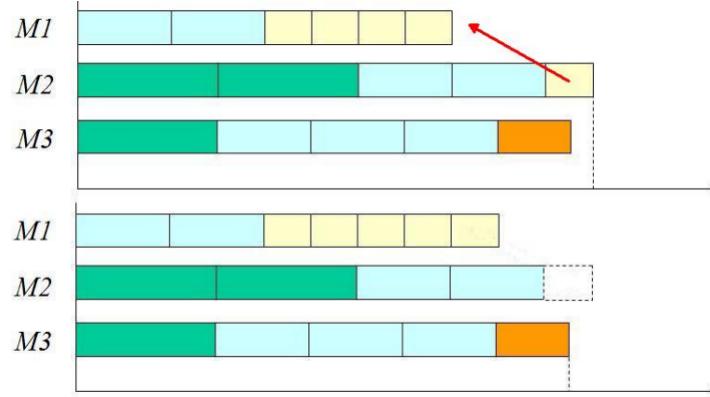


9 EURISTICHE DI SCAMBIO

Vicinato differente per lo stesso problema: PMSP

Per il PMSP è possibile definire:

- Il trasferimento dei vicinato $N_{\mathcal{T}_1}()$, basato sull'insieme \mathcal{T}_1 di tutti i trasferimenti di un task su un'altra macchina.



- lo scambio del vicinato $N_{\mathcal{S}_1}$, basato sull'insieme \mathcal{S}_1 di scambi di due task tra due macchine (uno per ogni macchina).

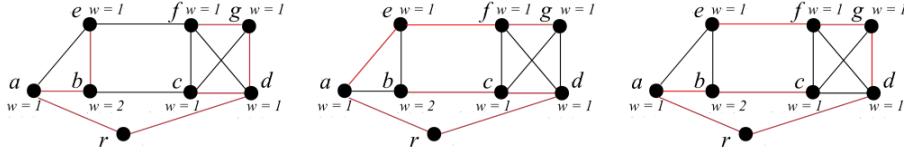
9.1.4 Connattività del grafo di ricerca

Un'euristica di scambio può restituire la soluzione ottima solo se tutte le soluzioni fattibili possono raggiungere almeno una soluzione ottimale, significa che è presente un cammino da x a X^* per ogni $x \in X$. Tale grafo di ricerca viene chiamato **debolmente connesso con l'ottimo** (*weakly connected to the optimum*). Visto che X^* è sconosciuto, una condizione più forte viene solitamente utilizzata: un grafo di ricerca è **fortemente connesso** quando ammette la presenza di cammini da x a $y \forall x, y \in X$. Un buon vicinato dovrebbe garantire alcune condizioni di connattività:

- Nel MDP, il vicinato $N\mathcal{S}_1$ connette qualsiasi coppia di soluzioni con al massimo k scambi.
- Nel KP e nel SCP, nessun vicinato $N\mathcal{S}_k$ fornisce una garanzia tale (soluzioni fattibili possono avere una qualsiasi cardinalità).
- Il grado di ricerca diventa connesso sia nel KP che nel SCP se le rimozioni e le aggiunte sono aggiunte a gli scambi.

Se la fattibilità è definita in una maniera sofisticata, scambiare, aggiungere ed eliminare **singoli** elementi potrebbe essere insufficiente per raggiungere tutte le soluzioni: i sottoinsiemi infatti possano rompere tutti i cammini tra alcune soluzioni fattibili.

9 EURISTICHE DI SCAMBIO



Nel caso in cui $V = 4$, solo tre soluzioni sono fattibili, tutte con due sottoalberi:

$$\begin{aligned} x &= \{(r,a), (a,b), (b,e), (r,d), (c,d), (d,g), (f,g)\} \\ x' &= \{(r,a), (a,e), (e,f), (f,g), (r,d), (c,d), (b,c)\} \\ x'' &= \{(r,a), (a,b), (b,c), (r,d), (d,g), (f,g), (e,f)\} \end{aligned}$$

Le tre soluzioni sono mutuamente raggiungibili solo se si scambiando due archi per volta; scambiando un singolo arco questo porta a sottoinsiemi insoddisfatti.

9.1.5 Euristica Hill-climbing

Il criterio di selezione più semplice $\varphi(x, A, D)$ è la funzione obiettivo. Viene utilizzata in quasi tutte le euristiche di scambio. Quando $\varphi(x, A, D) = d(x \cup A \setminus D)$, l'euroistica si muove da $x^{(t)}$ alla migliore soluzione in $N(x^{(t)})$. Per evitare comportamenti ciclici, solamente soluzioni che sono rigorosamente migliorative (*improving*) sono accettate. Conseguentemente, la migliore soluzione conosciuta è l'ultima visitata.

Algorithm 10 Algoritmo SteepestDescent($I, x^{(0)}$)

```

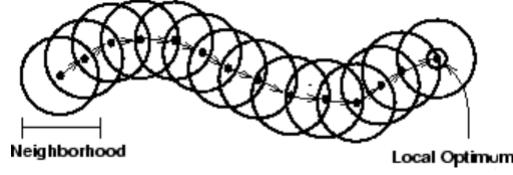
 $x := x^{(0)};$ 
 $Stop := false;$ 
while  $Stop == false$  do
     $\tilde{x} := \arg \min_{x' \in N(x)} f(x');$ 
    if  $f(\tilde{x}) \geq f(x)$  then
         $Stop := true;$ 
    else
         $x := \tilde{x};$ 
    end if
end while
Return  $(x, f(x));$ 
```

Ottimalità globale e locale

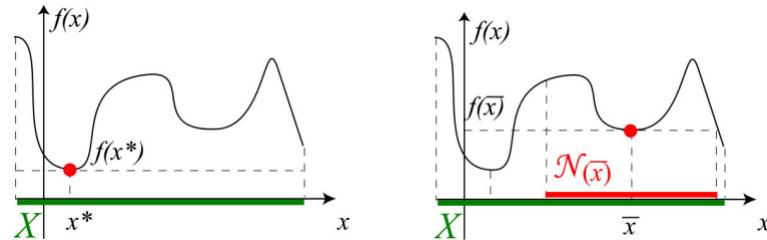
Un'euroistica *steepest descent* termina quando trova una soluzione ottima localmente, la quale è la soluzione $\bar{x} \in X$ tale che:

$$f(\bar{x}) \leq f(x) \text{ per ogni } x \in N(\bar{x})$$

9 EURISTICHE DI SCAMBIO



Una soluzione globalmente ottima è anche sempre localmente ottima, ma l'opposto non è vero in generale: $X^* \subseteq \bar{X}_N \subseteq X$



Vicinato esatto

Il vicinato esatto è una funzione "vicinato" $N : X \rightarrow 2^X$ tale che qualsiasi ottimo locale è anche un ottimo globale.

$$\bar{X}_N = X^*$$

Nel caso banale, il vicinato di ogni soluzione coincide con l'intera regione fattibile ($N(x) = X$ per ogni $x \in X$, esso è anche un vicinato troppo grande esplorare e quindi inutile). I vicinati **esatti** sono estremamente rari:

- scambio tra archi per il Minimum Spanning Tree Problem.
- scambio tra variabili base e non bases utilizzato da l'algoritmo simplex per la programmazione lineare.

In generale, l'euristica *steepest descent* non trova un ottimo globale. La sua efficacia dipende dalla proprietà del grafo di ricerca e dell'obiettivo.

9.1.6 Proprietà del grafo di ricerca

Alcune proprietà rilevanti per l'efficacia di un algoritmo sono:

- la dimensione dello spazio di ricerca $|X|$.
- la connettività del grafo di ricerca (precedentemente discussa).
- il diametro del grafo di ricerca, che è il numero di archi del cammino minimo tra le due soluzioni più distanti: vicinati più grandi producono grafi di diametri più piccoli.

9 EURISTICHE DI SCAMBIO

Considerando il vicinato N_{S_1} per il TSP simmetrico sui grafi completi:

- lo spazio di ricerca include $|X| = (n - 1)!$ soluzioni.
- N_{S_1} (scambio di due nodi) include $= \frac{n(n-1)}{2}$ soluzioni.
- il grafo di ricerca è fortemente connesso ed ha un diametro di $\leq n - 2$: tutte le soluzioni si trasformano in un'altra dopo al massimo $n/2$ scambi. Per esempio $x = (1, 5, 4, 2, 3)$ diventa $x' = (1, 2, 3, 4, 5)$ in 3 passi.

$$x = (1, 5, 4, 2, 3) \rightarrow (1, 2, 4, 5, 3) \rightarrow (1, 2, 3, 5, 4) \rightarrow (1, 2, 3, 4, 5) = x'$$

il primo nodo è sempre 1, l'ultimo è automaticamente in posizione.

Altre proprietà rilevanti:

- la densità del ottimo globale (*global optima density*, $\frac{|X^*|}{|X|}$) e dell'ottimo locale ($\frac{|\bar{X}_N|}{|X|}$). Se gli ottimi locali sono numerosi risulta difficile trovare l'ottimo globale.
- la qualità della distribuzione dell'ottimo locale $\delta(\bar{x})$ (diagramma SQD): se gli ottimi locali sono buoni, è meno importante trovare l'ottimo globale.
- la distribuzione delle soluzioni localmente ottime nello spazio di ricerca: se gli ottimi locali sono vicini tra di loro, non è necessario esplorare l'intero spazio.

Questi indici richiedono un'esplorazione esaustiva del grafo di ricerca. In pratica, eseguire un campionamento seguito dalle analisi comporta tempi molto lunghi e può essere ingannevole, specialmente se gli ottimi globali sono sconosciuti.

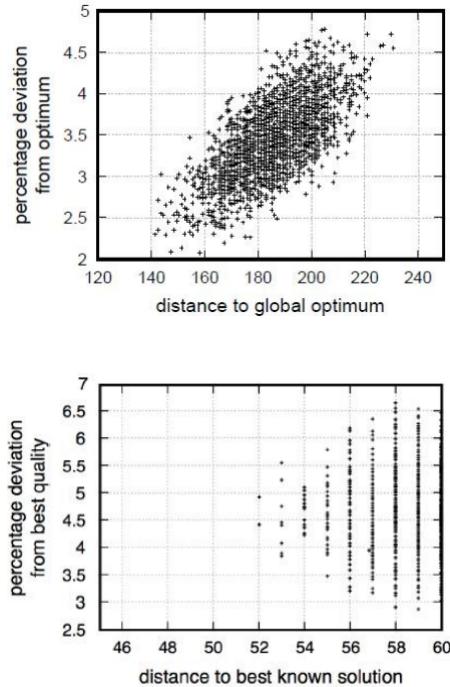
Per esempio il TSP

Per il TSP su un grafo completo simmetrico con costi Euclidei:

- la distanza di Hamming tra due ottimi locali è in media $\ll n$; l'ottimo locale si concentra in una piccola regione di X .
- la distanza di Hamming tra l'ottimo locale in media eccede quella tra l'ottimo locale e globale: l'ottimo globale tende a concentrare nel mezzo dell'ottimo locale.
- il diagramma FDC (*Fitness-Distance Correlation*) riporta la qualità $\delta(\bar{x})$ contro la distanza dall'ottimo globale $d_H(\bar{x}, X^*)$: se sono correlati, migliori ottimi locali sono più vicini a quelli globali.

Per il QAP (*Quadratic Assignment Problem*), la situazione è differente

9 EURISTICHE DI SCAMBIO



Se la qualità e la vicinanza all’ottimo globale sono fortemente correlate:

- è redditizio costruire delle buone soluzioni di partenza, perché guidano la ricerca vicino ad un buon ottimo locale.
- è meglio intensificare che diversificare.

Se la correlazione è debole:

- una buona inizializzazione è meno importante.
- è meglio diversificare che intensificare.

9.1.7 Il paesaggio

Il **landscape** o paesaggio è la tripla (X, N, f) , dove:

- X è lo spazio di ricerca, o l’insieme di soluzioni fattibili.
- $N : X \rightarrow 2^X$ è la funzione vicinato.
- $f : X \rightarrow \mathbb{N}$ è la funzione obiettivo.

è il grafo di ricerca con i nodi pesati dati dalla funzione obiettivo.

9 EURISTICHE DI SCAMBIO



L'efficacia della discesa più ripida (*steepest descent*) dipende dal paesaggio:

- landscape più lisci restituiscono pochi ottimi locali, possibilmente di buona qualità, e quindi buoni risultati.
- landscape più ruvidi restituiscono molti ottimi locali di qualità diffusa, quindi pessimi risultati.

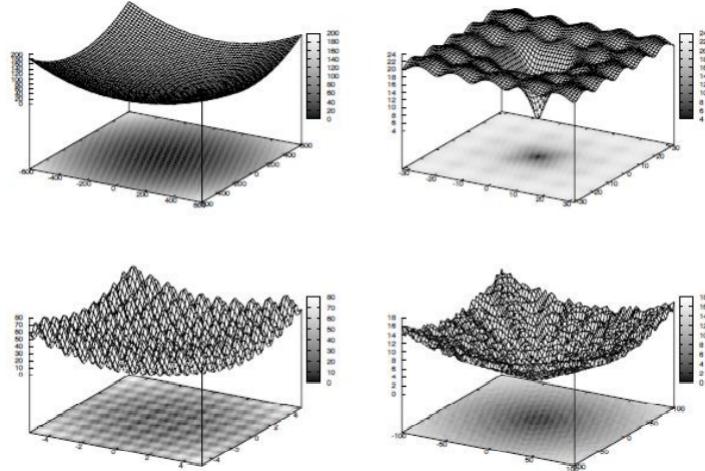


Figura 9.2: Sono presenti grandi varietà di landscape, molto differenti l'uno dall'altro

Coefficiente di autocorrelazione

La complessità di un paesaggio può essere empiricamente stimata:

1. eseguendo una camminata casuale nel grafo si ricerca.
2. determinando la sequenza di valori per la funzioni obiettivo $f^{(1)}, \dots, f^{(t_{max})}$
3. calcolando il campione medio $\bar{f} = \frac{\sum_{t=1}^{t_{max}} f(t)}{t_{max}}$
4. calcolando il coefficiente empirico di autocorrelazione:

$$r^{(i)} = \frac{\frac{\sum_{t=1}^{t_{max}-i} (f^{(t)} - \bar{f})(f^{(t+i)} - \bar{f})}{t_{max}-i}}{\frac{\sum_{t=1}^{t_{max}} (f^{(t)} - \bar{f})^2}{t_{max}}}$$

9 EURISTICHE DI SCAMBIO

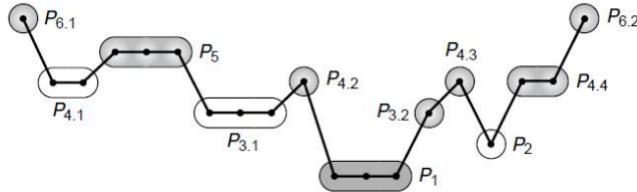
il quale mette in relazione la differenza dei valori obiettivo delle soluzioni visitate con la distanza tra queste soluzioni lungo il cammino.

Interpretazioni del valore:

- $r(0) = 1$, abbiamo una correlazione perfetta a distanza 0.
- generalmente $r(i)$ decresce al crescere della distanza i .
- se $r(i) \approx 1$ in un grande intervallo di distanze, la landscape è liscia (o *smooth*):
 - le soluzioni vicine hanno valori vicini a quella corrente.
 - ci sono pochi ottimi locali.
 - l'euristica *steepest descent* è efficace.
- se $r(i)$ varia notevolmente, allora il landscape sarà ruvido (o *rugged*):
 - le soluzioni vicine hanno valori distanti da quello corrente.
 - sono presenti molti ottimi locali.
 - l'euristica *steepest descent* non è efficace.

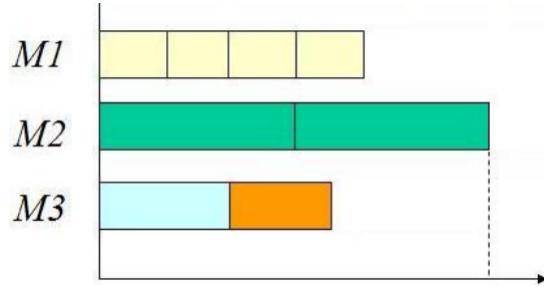
Plateau

Il grafo di ricerca può essere partizionato in accordo al valore della funzione obiettivo. Il **plateau** del valore di f è ogni sottoinsieme di soluzioni di valore f che sono adiacenti nel grafo di ricerca. Sostanzialmente il plateau è un sottoinsieme di soluzioni connesse che hanno lo stesso valore.



Per esempio, consideriamo la seguente istanza del PMSP, notiamo che se scambiamo i task tra la macchina 1 e 3 il risultato della funzione obiettivo non cambia.

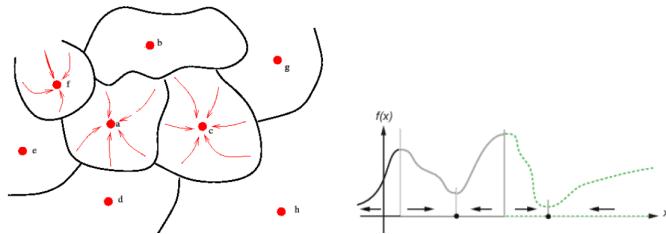
9 EURISTICHE DI SCAMBIO



Questo significa che non si avrà una buona euristica *steepest descent* (devi pensare a qualcos'altro).

Bacini di attrazione

Alternativamente, il grafo di ricerca può essere partizionato in **bacini di attrazione** (o *attraction basins*) delle soluzioni localmente ottime \bar{x} , che sono sottoinsiemi delle soluzioni $x^{(0)} \in X$ a partire dalle quali l'euristica *steepest descent* termina in \bar{x} .



nel caso in cui si abbia un piccolo numero di *grandi* bacini di attrazione allora probabilmente l'euristica *steepest descent* richiederà molti passi ma terminerà in una buona soluzione. Nel caso di significa che l'euristica farà pochi passi ma restituirà una brutta soluzione (meglio optare per un altro approccio). La *steepest descent* è:

- Efficace se i bacini di attrazione sono pochi e grandi.
- Inefficace se i bacini di attrazione sono tanti e piccoli.

9.2 Complessità Steepest Descent

Algorithm 11 Algoritmo *SteepestDescent*($I, x^{(0)}$)

```

 $x := x^{(0)};$ 
Stop := false;
while Stop == false do
     $\tilde{x} := \arg \min_{x' \in N(x)} f(x');$ 
    if  $f(\tilde{x}) \geq f(x)$  then
        Stop := true;
    else
         $x := \tilde{x}$ ;
    end if
end while
Return ( $x, f(x)$ );

```

La complessità dell'algoritmo di *steepest descent* dipende da:

1. Il numero di iterazioni t_{max} partendo da $x^{(0)}$ fino all'ottimo locale trovato, il quale dipende dalla struttura del grafo di ricerca (larghezza dei bacini di attrazione) ed è difficile da stimare *a priori*.
2. La ricerca della migliore soluzione nel vicinato \tilde{x} , la quale dipende da come la ricerca viene eseguita, solitamente richiede solo una stima delle complessità standard.

9.2.1 Esplorazione del vicinato

Sono presenti due strategie per esplorare il vicinato:

1. **ricerca esaustiva** (*exhaustive search*): si calcolano tutte le soluzioni dei vicini; la complessità di un singolo passo è il prodotto del:

- numero delle soluzioni dei vicini $|N(x)|$
- calcolo del costo di ogni soluzione $\gamma_f(|B|, x)$

Non è possibile generare solamente soluzioni fattibili:

- visita di un sovrainsieme del vicinato, $N(\tilde{x}) \supset N(x)$
 - per ogni elemento x , si calcola la fattibilità $\gamma_x(|B|, x)$
 - per le soluzioni fattibili si calcolano i costi $\gamma_f(|B|, x)$
2. **esplorazione efficiente del vicinato** senza una visita completa: si trova la migliore soluzione vicinato risolvendo un problema ausiliario (*solo dei vicini particolari lo permettono*).

Visita esaustiva del vicinato

Algorithm 12 Algoritmo *SteepestDescent*($I, x^{(0)}$)

```

 $x := x^{(0)};$ 
 $Stop := false;$ 
while  $Stop == false$  do
     $\tilde{x} = x;$ 
    for  $x' \in \tilde{N}(x)$  do
        if  $\tilde{x} \in N(x)$  then
            if  $f(x') < f(\tilde{x})$  then
                 $\tilde{x} := x';$ 
            end if
        end if
    end for
    if  $f(\tilde{x}) \geq f(x)$  then
         $Stop := true;$ 
    else
         $x := \tilde{x}$ 
    end if
end while
Return  $(x, f(x));$ 

```

Dove $\tilde{x} = \arg \min_{x' \in N(x)} f(x')$. La complessità del vicinato combina tre termini:

1. $|\tilde{N}(x)|$: il numero dei sottoinsiemi visitati.
2. γ_x : il tempo di calcolare la fattibilità.
3. γ_f : il temp di calcolare la funzione obiettivo di una soluzione fattibile.

9.2.2 Calcolare o aggiornare la funzione obiettivo: caso additivo

Il primo modo per accelerare un algoritmo di scambio è quello di minimizzare il tempo di valutazione della funzione obiettivo: in particolare è più veloce aggiornare $f(x)$ anziché ricalcularla.

L'aggiornamento di una funzione obiettivo $f(x) = \sum_{j \in x} \phi_j$ richiede di:

- sommare ϕ_i per ogni elemento di $i \in A$, aggiunto a x .
- sottrarre ϕ_j per ogni elemento di $j \in D$, rimosso da x

$$\delta f(x, A, D) = f(x \cup A \setminus D) - f(x) = \sum_{i \in A} \phi_i - \sum_{j \in D} \phi_j$$

Alcuni esempi: scambio di oggetti (KP), colonne (SCP), archi (CMSTP),...
Questo aggiornamento ha due proprietà fondamentali:

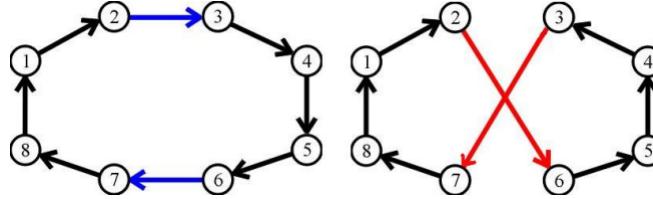
9 EURISTICHE DI SCAMBIO

- prende tempo costante per un numero costante di elementi $|A| + |D|$
- $\delta f(x, A, D)$ questo non dipende da x .

Esempio: il TSP simmetrico

Per generare il vicinato $N_{\mathcal{R}_2}$ per il TSP si deve:

- eliminare due archi non consecutivi (s_i, s_{i+1}) e (s_j, s_{j+1})
- aggiungere due archi (s_i, s_j) e (s_{i+1}, s_{j+1})
- invertire il percorso s_{i+1}, \dots, s_j (modificando $O(n)$ archi!).



Se il grafo e la funzione di costo sono simmetriche, la variazione di $f(x)$ è:

$$\delta f(x, A, D) = c_{s_i, s_j} + c_{s_{i+1}, s_{j+1}} - c_{s_i, s_{i+1}} - c_{s_j, s_{j+1}}$$

ma questo non è vero per il TSP asimmetrico. *Cosa se la funzione obiettivo non è additiva?*

9.2.3 Calcolare o aggiornare la funzione obiettivo: caso quadratico

Il MDP ha una funzione obiettivo quadratica, il costo di computazione è $\Theta(n^2)$. Spostandoci da x a $x' = x \setminus \{i\} \cup \{j\}$ (vicinato $N_{\mathcal{S}_1}$), l'aggiornamento è:

$$\delta f(x, i, j) = f(x \setminus \{i\} \cup \{j\}) - f(x) = \sum_{h, k \in x \setminus \{i\} \cup \{j\}} d_{hk} - \sum_{h, k \in x} d_{hk}$$

che dipende da $O(n)$ termini di distanza relativi ai punti i e j . È presente un trucco generale per le funzioni simmetriche quadratiche con $d_{ii} = 0$

$$\begin{aligned} \delta f(x, i, j) &= \sum_{h \in x \setminus \{i\} \cup \{j\}} \sum_{k \in x \setminus \{i\} \cup \{j\}} d_{hk} - \sum_{h \in x} \sum_{k \in x} d_{hk} \implies \\ \implies \delta f(x, i, j) &= 2 \sum_{k \in x} d_{jk} - 2 \sum_{k \in x} d_{ik} - 2d_{ij} = \\ &= 2(D_j(x) - D_i(x) - d_{ij}) \end{aligned}$$

Se $D_\ell(x) = \sum_{k \in x} d_{\ell k}$ è conosciuto per ogni $\ell \in B$, la computazione richiede $O(1)$.

9 EURISTICHE DI SCAMBIO

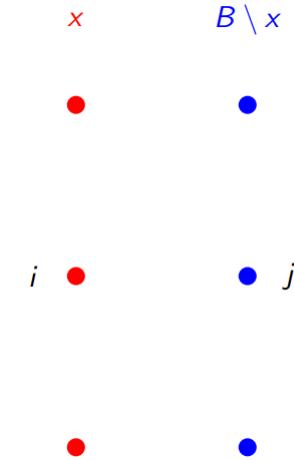
Esempio: il MDP

Consideriamo $f(x)/2$, calcoliamo lo scambio:

$$x \rightarrow x' = x \setminus \{i\} \cup \{j\}$$

con $i \in x$ e $j \in B \setminus x$

$$f(x') = f(x) - D_i + D_j - d_{ij}$$



- le coppie che includono i sono perse.
- le coppie che includono j sono acquisite.
- ma la coppia (i, j) è in eccesso.

Il costo è calcolato in tempo $O(1)$ per ogni soluzione.

Aggiornamento delle strutture dati:

$$D_\ell = D_\ell - d_{\ell i} + d_{\ell j}, \ell \in B$$

Per ogni elemento $\ell \in B$:

- $d_{\ell i}$ sparisce.
- $d_{\ell j}$ compare.

La struttura dati ausiliare è aggiornata in tempo $O(n)$ per ogni iterazione.

9.2.4 Aggironamento della funzione obiettivo: esempi non lineari

Molte funzioni non-lineari possono essere aggiornate con simili trucchi:

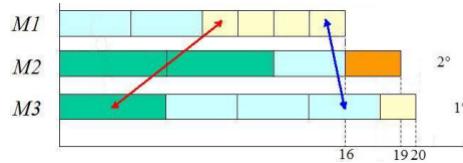
- salva le informazioni aggregate sulla soluzione corrente $x^{(t)}$.
- utilizzale per calcolare $f(x')$ efficientemente per ogni $x' \in N(x^{(t)})$.
- aggiornalo quando passi alla seguente soluzione x^{t+1} .

9 EURISTICHE DI SCAMBIO

Esempio: PMSP

Utilizzare il trasferimento (N_{T_1}) e lo scambio (N_{S_1}) del vicinato per il PMSP, la funzione obiettivo può essere aggiornata in tempo costante, questo gestendo:

- il tempo di completamento per ogni macchina.
- gli indici della macchina con il primo e secondo tempo massimo.



Considera lo scambio $o = (i, j)$ dei task i e j (rispettivamente i sulla macchina M_i , j sulla macchina M_j).

- calcolare in tempo costante i nuovi tempi di completamento: uno incrementa, l'altro decrementa (o entrambi rimangono costanti).
- prova (in tempo costante) se uno dei due supera il massimo.
- se il tempo massimo decresce, provare (in tempo costante) quando l'altro tempo o il secondo tempo massimo diventa il massimo.

Una volta che il vicinato è visitato e lo scambio selezionato, aggiorna:

- i due tempi di completamento modificati (entrambi in tempo costante).
- le loro posizioni in un max-heap (ogni uno in tempo $O(\log |M|)$)

Esempio: TSP asimmetrico

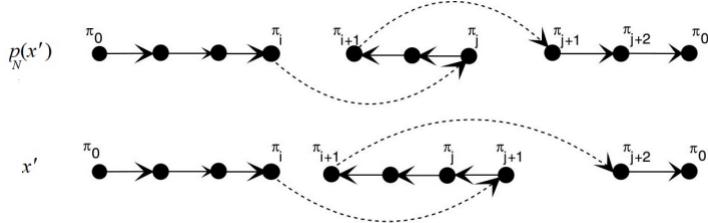
L'utilizzo dell'informazione ausiliaria utilizzata per il calcolo di $f'(x)$ può essere:

- **globale**, il quale si riferisce alla corrente soluzione x .
- **locale**, il quale si riferisce alla soluzione $p_N(x')$ visitata precedentemente nel vicinato $N(x)$ secondo un opportuno ordine.

Consideriamo il vicinato N_{R_2} per il TSP asimmetrico:

- le soluzioni del vicinato differiscono da x per $O(n)$ archi.
- le soluzioni del vicinato generali differiscono una dall'altra per $O(n)$ archi.
- se la coppia di archi (s_i, s_{i+1}) e (s_j, s_{j+1}) segue l'ordine lessicografico, il cammino inverso cambia solo rispetto ad un arco.

9 EURISTICHE DI SCAMBIO



La variazione di $f(x)$ tra due soluzioni vicinato generiche è:

$$\delta f(x, i, j) = c_{s_i, s_j} + c_{s_{i+1}, s_{j+1}} - c_{s_i, s_{i+1}} - c_{s_j, s_{j+1}} + c_{s_j, \dots, s_{i+1}} - c_{s_{i+1}, \dots, s_j}$$

ma muoversi dallo scambio (s_i, s_j) allo scambio (s_i, s_{j+1}) :

- i primi quattro termini cambiano, ma possono essere controllati in tempo costante.
- gli ultimi due termini possono essere aggiornati in tempo costante

$$\begin{cases} c_{s_{j'} \dots s_{i+1}} = c_{s_j \dots s_{i+1}} + c_{s_{j+1}, s_j} \\ c_{s_{i+1} \dots s_{j'}} = c_{s_{i+1} \dots s_j} + c_{s_j, s_{j+1}} \end{cases}$$

Cosa a riguardo della fattibilità? Definire vicinati con una distanza di Hamming o con operazioni può generare anche sottoinsiemi infattibili, i quali devono essere rimossi.

$$\tilde{N}_H(x) = \{x' \subseteq B : d(x', x) \leq k\} \supseteq N_{\mathcal{O}}(x) = \tilde{N}_H(x) \cap X$$

$$\tilde{N}_H(x) = \{x' \subseteq B : \exists o \in \mathcal{O} : o(x) = x'\} \supseteq N_{\mathcal{O}}(x) = \tilde{N}_{\mathcal{O}}(x) \cap X$$

Per esempio: KP, BPP, SCP, CMSTP, ... Se non è possibile evitare *a priori* i sottoinsiemi infattibili, si deve:

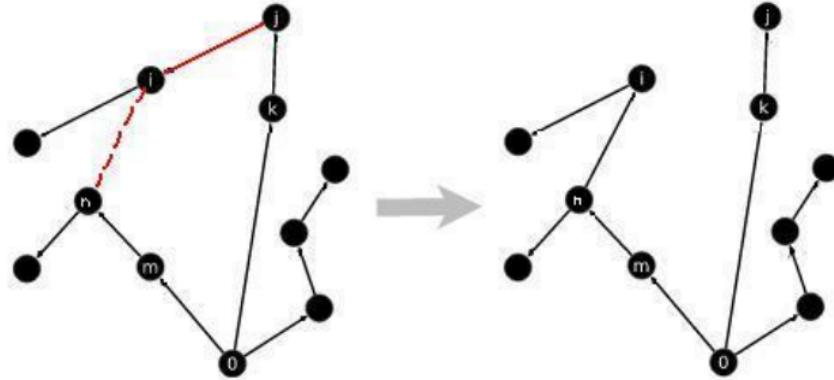
- provare/testare la fattibilità di ogni elemento di $\tilde{N}(x)$ per ottenere $N(x)$.
- per gli elementi fattibili, calcolare il costo.

Il test di fattibilità può essere reso efficiente con delle tecniche simili a quelle utilizzate per la valutazione della funzione obiettivo. Un esempio di questo potrebbe essere aggiornare in tempo costante il volume totale di un sottoinsieme nel KP.

Esempio: CMSTP

Consideriamo il vicino di scambio N_{S_1} (aggiungo un arco e rimuovo un altro).

- se i due archi sono nello stesso ramo, la soluzione rimane fattibile.
- se loro sono in rami differenti, uno perde peso, l'altro guadagna peso: la variazione è uguale al peso del sotto-albero trasferito.



Se ogni vertice slava il peso del proprio sotto-albero accodato, nel testare la fattibilità si confronta questo peso con la capacità residua del ramo ricevente (il peso accodato a i con la capacità residua del ramo sinistro).

Una volta che lo scambio migliore stato effettuato, l'informazione deve essere aggiornata in tempo $O(n)$ visitando i predecessori del vertice ricevente h .

9.2.5 Schema generale di esplorazione sofisticata

L'utilizzo dell'informazione ausiliaria richiede:

1. L'inizializzazione di strutture dati adatte
 - In parte locali, sostanzialmente quelle relative alle soluzioni vicine.
 - In parte globali, sostanzialmente quelle relative alle soluzioni correnti.
2. il loro aggiornamento tra le soluzioni e iterazioni successive.

Algorithm 13 Algoritmo *SteepestDescent*($I, x^{(0)}$)

```

 $x := x^{(0)};$ 
 $GI := InitializeGI();$ 
 $Stop := false;$ 
while  $Stop == false$  do
     $\tilde{x} = 0;$ 
     $\tilde{\delta} := 0;$ 
     $LI := InitializeLI();$ 
    for  $x' \in \tilde{N}(x)$  do
        if  $\tilde{x} \in N(x)$  then
            if  $f(x') < f(\tilde{x})$  then
                 $\tilde{x} := x';$ 
                 $LI := UpdateLI(LI, x')$ 
            end if
        end if
    end for
    if  $f(\tilde{x}) \geq f(x)$  then
         $Stop := true;$ 
    else
         $x := \tilde{x}$ 
         $GI := UpdateGD(GI, \tilde{x})$ 
    end if
end while
Return  $(x, f(x));$ 

```

9.2.6 Salvataggio parziale del vicinato

Quando si sta eseguendo un operazione $o \in \mathcal{O}$ su una soluzione $x \in X$ la variazione

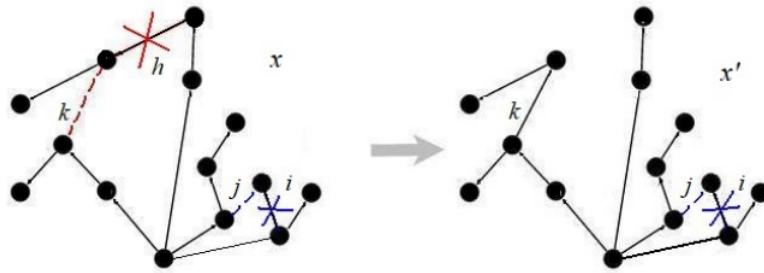
$$\delta f_o(x) = f(o(x)) - f(x)$$

a volte dipende solo su una parte di x , possibilmente molto piccola. Per esempio, consideriamo il vicinato di scambio N_{S_1} per il CMST:

- aggiunge un arco $k \in B \setminus x$
- rimuovi un arco $h \in x$

Due rami sono coinvolti: uno acquisisce un sotto-albero, l'altro lo perde

9 EURISTICHE DI SCAMBIO



L'effetto dello scambio (i, j) dipende solo sui rami inclusi i e j : è lo stesso in x e x' e non è influenzato dallo swap (h, k) .

$$\delta f_{i,j}(x) = \delta f_{i,j}(x')$$

Esiste un sottoinsieme delle operazioni $\tilde{\mathcal{O}} \subset \mathcal{O}$ tale che:

$$\delta f_o(x') = \delta f_o(x) \text{ per ogni } o \in \tilde{\mathcal{O}} \text{ e per ogni } x' = o(x)$$

Allora risulta vantaggioso:

1. calcolare e salvare $\delta f_o(x)$ per ogni $o \in \mathcal{O}$, ovvero mantenere l'insieme degli scambi ammissibili e i loro valori associati δf .
2. eseguire la migliore operazione o^* , e generare una nuova soluzione x' .
3. ricalcolare e salvare $\delta f_o(x')$ solo per $o \in \mathcal{O} \setminus \tilde{\mathcal{O}}$, ovvero rimuovere gli scambi sui rami modificati, ricalcolare i loro valori e recuperare $\delta f_o(x')$ per tutti $o \in \tilde{\mathcal{O}}$ (i loro valori sono ancora corretti).
4. tornare al punto 2.

Nel caso in cui i rami siano numerosi, $|\mathcal{O} \setminus \tilde{\mathcal{O}}| \subset |\mathcal{O}|$ e il risparmio è molto forte. Questo è tipico dei problemi la cui soluzione è una partizione.

9.2.7 Trade-off tra efficienza ed efficacia

La complessità di un'euristica di scambio dipende da tre fattori:

1. numero di iterazioni.
2. cardinalità del vicinato visitato.
3. calcolo della fattibilità e costi per un singolo vicino.

I primi due fattori sono chiaramente in conflitto:

- un piccolo vicinato è veloce da esplorare, ma richiede diversi step prima di raggiungere un ottimo locale.
- un grande vicinato richiede pochi passi, ma è molto lento da esplorare.

10 RICERCA DI VICINATO SU LARGA SCALA

Lo scambio (*trade-off*) ottimale si trova a metà: un vicinato tale che:

- sia grande abbastanza da includere le buone soluzioni.
- piccolo abbastanza da essere esplorato velocemente.

ma è difficile da identificare, perché:

- l'efficienza peggiora in fretta con l'aumentare della dimensione.
- la soluzione risultante cambia con il vicinato (grandi vicinati hanno migliori ottimi locali).

9.2.8 Ritocchi dei vicinati

Risulta possibile definire un vicinato N e regolare la sua dimensione:

- esplorare solo un promettente sotto-vicinato $N' \subset N$. Per esempio, se la funzione obiettivo è additiva, uno può:
 - aggiungere solo gli elementi $j \in B \setminus x$ di basso costo ϕ_j .
 - rimuovere solo gli elementi $i \in x$ di alto costo ϕ_i
- terminare la visita dopo aver trovare una soluzione promettente. Per esempio, la strategia *first-best* che termina l'esplorazione alla prima soluzione migliore rispetto a quella corrente:

```
if  $f(\tilde{x}) < f(x)$  then  $x := \tilde{x}$ ;  $Stop := true$ ;
```

L'efficacia dipende dall'obiettivo:

- se il costo di alcuni elementi influenza molto l'obiettivo, vale la pena tenerne conto, fissandone il divieto.

sulla struttura del quartiere:

- se il paesaggio è tranquillo, la prima soluzione migliorativa si avvicina (approssima) bene alla migliore soluzione del vicinato: è meglio fermarsi.
- se il paesaggio è rugoso, la soluzione migliore del vicinato potrebbe essere molto migliore: è meglio proseguire.

10 Ricerca di vicinato su larga scala

Grandi vicinati forniscono, in generale, grandi bacini di attrazione, tali per cui:

- l'euristica *steepest descent* diventa molto efficace.
- i tempi di esplorazione si allungano.

10 RICERCA DI VICINATO SU LARGA SCALA

Gli approcci *Very Large Scale Neighbourhood Search* (VLSN Search) hanno:

- vicinati esponenziali in $|B|$ (o polinomi di ordine più alto).
- esplorazione in tempo polinomiale di basso-ordine.

Esistono due strategie per limitare il tempo computazionale:

1. selezionare un vicinato in cui l'obiettivo può essere ottimizzato senza un'esplorazione esaustiva.
2. esplorare il quartiere in modo euristico e restituire una soluzione vicinato promettente, invece della migliore.

10.1 Visita efficiente di vicinati esponenziali

Alcuni metodi sono basati su vicinati la cui soluzione ottimale può essere trovata risolvendo un problema su una matrice ausiliaria o grafo.

- set packing: *Dynasearch*
- negative cost circuit: *cyclic exchanges*
- shortest path: *ejection chains, order-and-split*

Tali strumenti sono solitamente definiti come matrici di miglioramento (*improvement matrix*) o grafi.

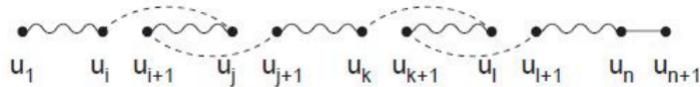
10.1.1 Dynasearch

La variazione $\delta f_o(x)$ della funzione obiettivo implicata da un'operazione $o \in \mathcal{O}$ spesso dipende solo da una parte della soluzione x . Le operazioni che modificano la stessa parte sono generalmente incompatibili. Le operazioni che modificano parti differenti hanno effetti indipendenti e l'ordine in cui sono eseguite è irrilevante.

$$f(o'(o(x))) = f(o(o'(x)))$$

Se $f(\cdot)$ è additiva, i due scambi semplicemente sommano i loro effetti: Esempi:

- gli scambi tra differenti rami per il CMSTP o differenti circuiti per il VRP (altrimenti, lo scambio potrebbe essere infattibile).
- 2-opt scambi per il TSP operanti sui segmenti disgiunti del circuito (altrimenti, uno dei due archi è rimosso o invertito).



10 RICERCA DI VICINATO SU LARGA SCALA

Un intero insieme di mosse può essere eseguito assieme sommandone i loro effetti. Sia una **mossa composta** un insieme di mosse elementari che hanno degli effetti mutuamente indipendenti sulla fattibilità e sull'obiettivo.

La situazione può essere modellata con una matrice di miglioramento A nella quale:

- le righe rappresentano i componenti della soluzione (es., i rami del CMSTP, circuiti del VRP, segmenti di circuito nel TSP).
- le colonne rappresentano le mosse elementari: ogni colonna ha un valore uguale al miglioramento obiettivo $-\delta f$.
- se la mossa j impatta sul componente i , $a_{ij} = 1$, altrimenti $a_{ij} = 0$.

Determinare l'impacchettamento ottimale delle colonne, ovvero il sottoinsieme di colonne non in conflitto di valore massimo. Il *Set Packing Problem* è generalmente un problema \mathcal{NP} -hard, ma:

- su speciali matrici risulta polinomiale (come nella matrice del TSP).
- se ogni mossa modifica al massimo due componenti:
 - le righe possono essere viste come nodi.
 - le colonne possono essere viste come archi.
 - ogni raggruppamento delle colonne diventa un matching.

ed il problema del massimo matching è polinomiale.

10.1.2 Scambi ciclici

In molti problemi:

- ogni soluzione ammissibile (*fattibile*) partiziona gli oggetti in componenti $S^{(\ell)}$: consiste in assegnazioni di oggetti a componenti (i, S_i) (vertici o archi in rami per il CMSTP, nodi o archi nei circuiti per il VRP, oggetti in container nel BPP, ...).
- l'ammissibilità è associata ai singoli componenti.
- la funzione obiettivo è additiva rispetto alle componenti.

$$f(x) = \sum_{\ell=1}^r f(s^{(\ell)})$$

Risulta naturale definire per questi problemi l'insieme delle operazioni T_k le quali includono i trasferimenti di k elementi dai loro componenti ad altri e derivare da T_k il vicinato N_{T_k} .

- spesso i vincoli di fattibilità vietano i semplici trasferimenti.
- ma il numero di trasferimenti multipli cresce velocemente con k .

Vogliamo trovare un sottoinsieme di N_{T_k} grande, ma efficiente da esplorare.

10.1.3 Grafo di miglioramento

Il grafo di miglioramento permette di descrivere sequenze di trasferimenti.

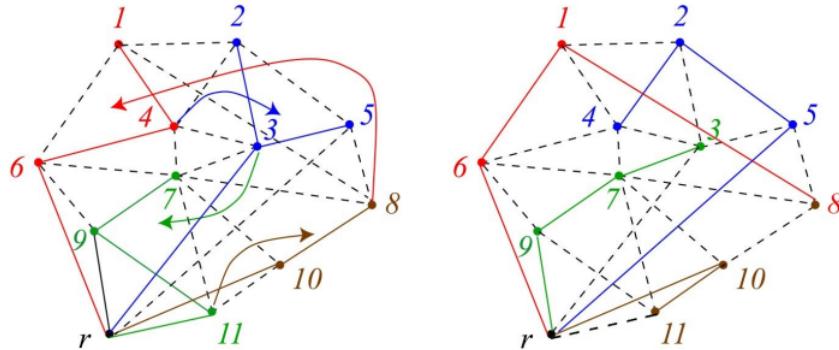
- un nodo i corrisponde ad un elemento i della soluzione corrente x .
- un arco (i, j) corrisponde a:
 - il trasferimento dell'elemento i dalla sua componente corrente S_i alla componente corrente S_j dell'elemento j .
 - l'eliminazione dell'elemento j dal componente S_j .
- il costo dell'arco c_{ij} corrisponde alla variazione del contributo di S_j all'obiettivo:

$$c_{ij} = f(S_j \cup \{i\} \setminus \{j\}) - f(S_j)$$

con $c_{ij} = +\infty$ se è infattibile trasferire i e rimuovere j .

Un circuito in un grafo tale corrisponde ad una sequenza chiusa di trasferimenti. Il costo del circuito corrisponde al costo della sequenza, ma solo se ogni nodo appartiene ad una componente diversa. Trovare il costo minimo del circuito soddisfa questa condizione.

Esempio: il CMSTP



Consideriamo la mossa composta $(4, 3), (3, 11), (11, 8), (8, 4)$:

- il vertice 4 si muove nel ramo blu per sostituire il vertice 3.
- il vertice 3 si muove nel ramo verde per sostituire il vertice 11.
- il vertice 11 si muove nel ramo marrone per sostituire il vertice 8.
- il vertice 8 si muove nel ramo rosso per sostituire il vertice 4.
- Il costo della variazione per il sotto-albero S_j restituisce il costo dell'arco c_{ij} .
- il peso del ramo S_j varia di $w_i - w_j$: se non fattibile, vietare l'arco.

10.1.4 Ricerca per il circuito di costo minimo

Il problema è attualmente \mathcal{NP} -hard, ma:

- il vincolo di visitare una sola volta ogni componente consente un algoritmo di *programmazione dinamica* piuttosto efficiente che fa crescere cammini parziali (*se le componenti sono r , il circuito ha al massimo r archi*).
- tutti i cammini parziali di costo ≥ 0 possono essere negativi perché:
 - la variazione totale della funzione obiettivo somma gli effetti di tutte le singole mosse

$$\delta f_{o_1, \dots, o_k}(x) = \sum_{\ell=1}^k \delta f_{o_\ell}(x)$$

- ogni sequenza di numeri con somma negativa ammette una permutazione ciclica le cui somme parziali sono tutte negative.
- quindi, è presente una permutazione ciclica delle mosse o_1, \dots, o_k con $\delta f < 0$ ad ogni passo

$$\delta f_{o_1, \dots, o_k}(x) < 0 \implies \exists h : \sum_{\ell=1}^{\ell} \delta f_{o_{h+1}, \dots, o_{h+\ell}}(x) < 0 \quad \forall \ell = 1, \dots, k$$

Inoltre:

- il problema ammette algoritmi euristici polinomiali.
- sono presenti algoritmi polinomiali da calcolare:
 - un circuito negativo non-minimo (Floyd-Warshall), se esistente.
 - il circuito minimo di costo medio (costo totale / numero di archi).

rilassare il vincolo sulle componenti; se tali soluzioni hanno:

- costo positivo, dimostrano che non esiste alcun circuito negativo.
- costo negativo, forniscono soluzioni rilassate che possono essere
 - * ottimali (se fattibile per caso).
 - * potenzialmente modificate per ottenere soluzioni euristiche fattibili.

10.1.5 Catene di scambio non cicliche

È anche possibile creare catene di trasferimento non cicliche, in modo che la cardinalità dei componenti possa variare.

Questo è abbastanza per aggiungere al grafo di miglioramento:

- un nodo sorgente.

10 RICERCA DI VICINATO SU LARGA SCALA

- un nodo per ogni componente.
- archi dal nodo sorgente ai nodi associati agli elementi.
- archi dai nodi associati a gli elementi ai nodi associati alle componenti.

Dopo, trovare il *cammino di costo minimo* che:

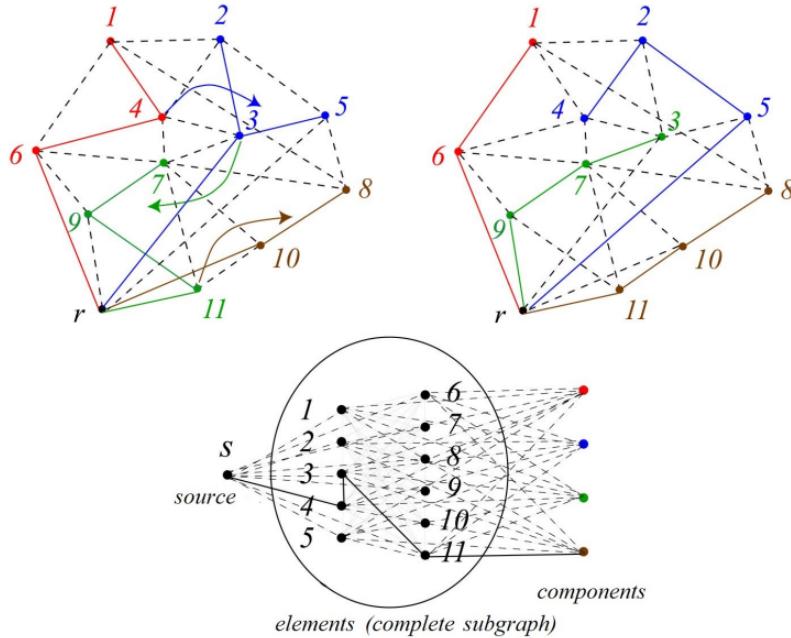
- inizia dal nodo sorgente.
- termina in un nodo componente (garantito dalla topologia).
- non visita mai due nodi associati allo stesso componente.

Questi cammini corrispondono a catene di trasferimento aperte nelle quali:

- una componente perde un elemento.
- zero o più componenti perdono un elemento e ne acquisiscono un altro.
- un componente acquisisce un elemento.

Esempio: il CMSTP

Scambi non ciclici: $(s, 4)$, $(4, 3)$, $(3, 11)$, $(11, S_4)$



10.1.6 Order-first split-second

Il metodo *Order-first split-second* per la partizione dei problemi:

- inizialmente costruisce una permutazione iniziale degli elementi.
- suddivide gli elementi in componenti in modo ottimale sotto il vincolo addizionale che gli elementi dello stesso componente siano consecutivi nella permutazione di partenza.

Certamente, la soluzione dipende dalla permutazione iniziale: è ragionevole ripetere la risoluzione per le differenti permutazioni, creando un metodo a due livelli:

1. il livello in alto seleziona una permutazione.
2. il livello in basso calcola la partizione ottimale per la permutazione.

Problema: differenti permutazioni restituiscono la stessa soluzioni (*le permutazioni sono più numerose delle soluzioni*).

10.1.7 Grafo ausiliario

Ancora una volta, utilizziamo un grafo ausiliario, data la permutazione (s_1, \dots, s_n) degli elementi:

- ogni nodo s_i corrisponde ad un elemento s_i più un nodo fittizio s_0 .
- ogni arco (s_i, s_j) con $i < j$ corrisponde ad un potenziale componente $S_\ell = (s_{i+1}, \dots, s_j)$ formata dagli elementi della permutazione:
 - da s_i escluso.
 - a s_j inclusa.
- il costo c_{s_i, s_j} corrisponde al costo del componente $f(S_\ell)$
- l'arco non esiste se il componente è infattibile.

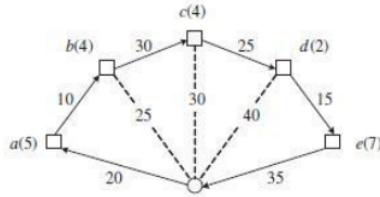
Conseguentemente:

- ogni cammino da s_0 a s_n rappresenta una soluzione (partizione di elementi).
- il costo di un cammino coincide con il costo della partizione.
- Il grafo è aciclico: trovare il cammino ottimale costa $O(m)$ dove $m \leq n(n - 1)/2$ è il numero di archi.

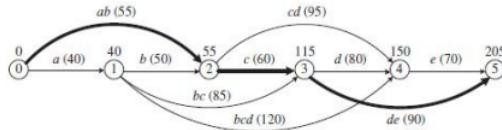
10 RICERCA DI VICINATO SU LARGA SCALA

Esempio: il VRP

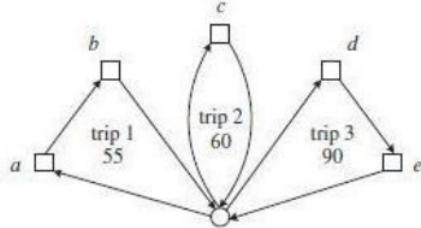
Data un'istanza del VRP con 5 nodi e la capacità $W = 10$



gli archi corrispondente ai cammini infattibili ($\text{peso} > W$) non esistono, il costo degli archi sono i costi delle soluzioni TSP per $\{d, s_{i+1}, \dots, s_j\}$.



Il cammino ottimale corrisponde ai tre circuiti: (d, s_1, s_2, d) , (d, s_3, d) e (d, s_4, s_5, d) .



10.1.8 Ricerca in profondità variabile

La *Variabile Depth Search* (VDS). I vicinati basati sulle operazioni che possono essere facilmente parametrizzate

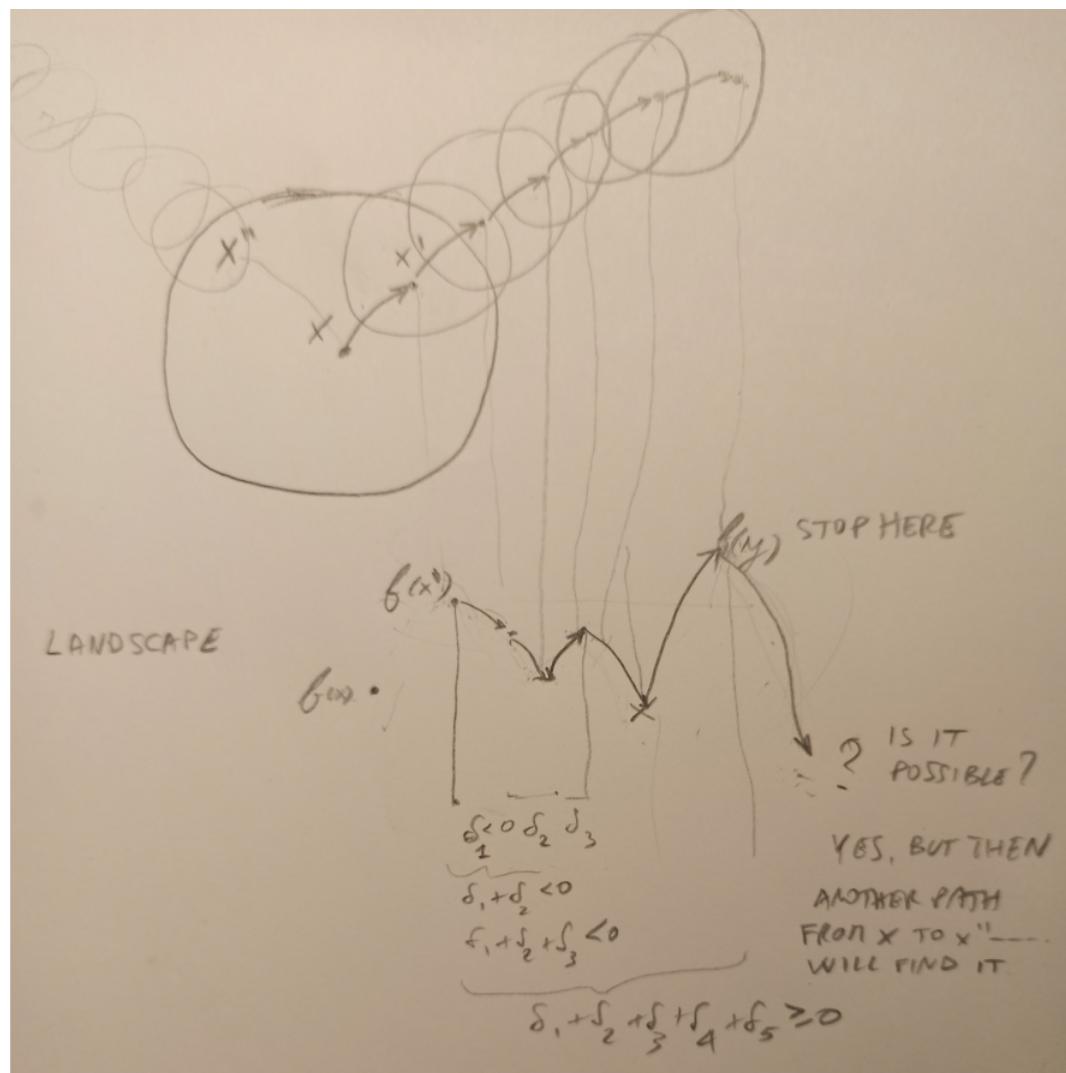
$$N_{\mathcal{O}_k}(x) = \{x' \in X : x' = o_k(o_{k-1}(\dots o_1(x))) \text{ con } o_1, \dots, o_k \in \mathcal{O}\}$$

sarebbe bello regolare il numero di operazioni k come richiesto. L'idea è quella di definire una mossa composta come una sequenza di mosse elementari:

- considerare ogni soluzione x' nel vicinato di base $N_{\mathcal{O}_1}(x)$.
- da esso, fare una sequenza di mosse ottimizzando ogni passo elementare, ma consentendo mosse peggiori e vietando le mosse all'indietro.
- terminano quando la soluzione corrente y diventa peggiore di x' o tutte le mosse sono proibite (la lunghezza k della sequenza è variabile).

10 RICERCA DI VICINATO SU LARGA SCALA

- restituire la soluzione migliore y^* trovata lungo la sequenza.



Schema del VDS

Dato $x^{(t)}$, per ogni $x' \in N(x^{(t)})$, anziché calcolare solamente $f(x')$:

1. trovare una soluzione promettente \tilde{y} in un vicinato $\hat{N}(x') \subseteq N(x')$
2. finché \tilde{y} migliora $x^{(t)}$, sostituisco x' con \tilde{y} e vado al passo 1.
3. restituisco la soluzione migliore y^* trovata durante l'intero processo.

Per ogni $x' \in N(x)$

Algorithm 14 *Steepest descent*

Compute $f(x')$

Algorithm 15 *Variable Depth Search*

```

 $y := x';$ 
 $y^* := x;$ 
 $Stop := false;$ 
while  $Stop == false$  do
     $\tilde{y} = \arg \min_{y' \in \hat{N}(y)} f(y');$ 
    if  $f(\tilde{y}) \geq f(x')$ 
         $Stop := true;$ 
    else
         $y := \tilde{y};$ 
    end if
    if  $f(\tilde{y}) < f(y^*)$ 
         $y^* := \tilde{y};$ 
    end if
end while
Compute  $f(y^*)$ ;

```

Differenze rispetto a *steepest descent*

Rispetto all'esplorazione *steepest descent*:

- *VDS* trova l'ottimo locale per ogni soluzione del vicinato eseguendo una sorta di "look-ahead" di un passo.
- *VDS* ammette peggioramenti lungo la sequenza delle mosse elementari (ma mai rispetto alla soluzione di partenza).
- *VDS* effettua spostamenti che aumentano la distanza dal punto di partenza per evitare comportamenti ciclici (limitando progressivamente il vicinato).

Per limitare lo sforzo computazionale:

- le mosse elementari utilizzano un vicinato ridotto $\hat{N} \subseteq N$.
- \hat{N} (passi elementari) viene esplorato con la strategia *first-best*.
- N (vicinati base) viene esplorato con la strategia *first-best*.

10.1.9 Algoritmo Lin-Kernighan (TSP simmetrico)

Il vicinato $N_{\mathcal{R}_k}$ include le soluzioni ottenute:

- rimuovendo k archi da x .
- aggiungendo altri k archi per ricreare un circuito Hamiltoniano.
- possibilmente invertendo parti del circuito (*lasciando il costo invariato*).

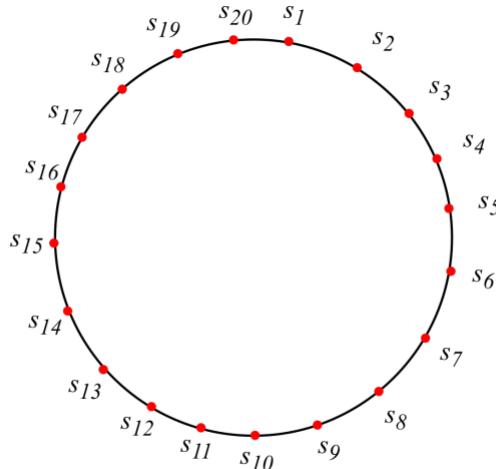
L'algoritmo di Lin-Kernighan è un VDS con sequenze di 2-opt scambi: uno scambio k -opt è equivalente ad una sequenza di $(k - 1)$ scambi 2-opt, dove ciascuno cancella uno dei due archi aggiunto dallo scambio precedente.

Dopo per ogni soluzione $x' \in N_{\mathcal{R}_2}(x)$ ottenuto scambiando (i, j) :

- calcola gli scambi 2-op che rimuovono l'arco aggiunto (s_i, s_{j+1}) e ogni arco $x \cap x'$ per trovare lo scambio migliore (i', j') .
- se questo migliora su x , effettua lo scambio (i', j') , ottenendo x'' .
- calcolare gli scambi che rimuovono $(s_{i'}, s_{j'+1})$ e ogni arco di $x \cap x'' \dots$
- ...
- se la soluzione migliore tra x', x'', \dots è migliore di x , accettala.

Esempio: Algoritmo Lin-Kernighan

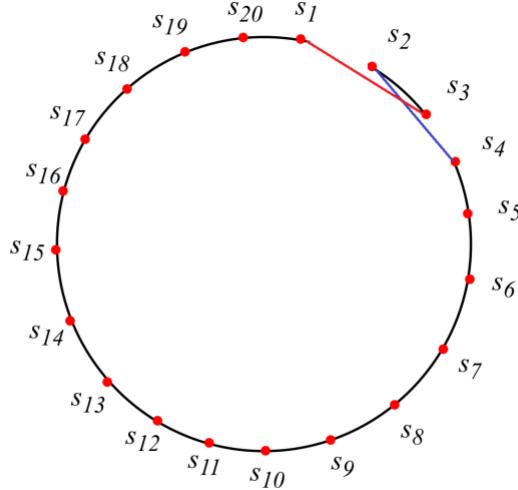
Esplora tutte le soluzioni $x' \in N_{\mathcal{R}_2}(x)$, ottenuto con gli scambi (i, j) .



$$x = (s_1 \overline{s_2} \overline{s_3} s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20})$$

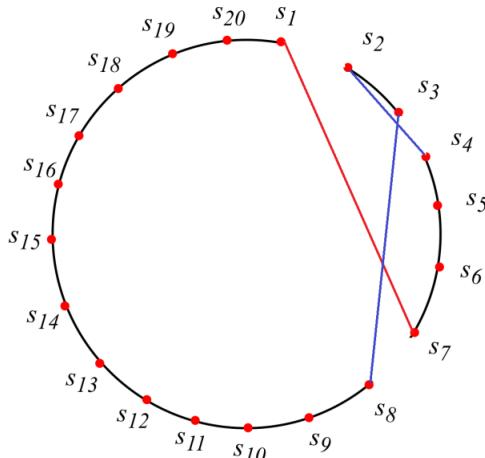
Concentriamoci sullo scambio $(1, 3)$, che sostituisce (s_2, \dots, s_3) . Questo scambio sostituisce (s_1, s_2) e (s_3, s_4) con (s_1, s_3) e (s_2, s_4) .

10 RICERCA DI VICINATO SU LARGA SCALA



$$x' = (s_1 \overline{s_3} s_2 s_4 s_5 s_6 \overline{s_7} s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20})$$

Cerca lo scambio migliore che rimuove (s_1, s_3) e un arco $x \cap x'$. Assumiamo che questo scambio sia $(1, 7)$, il quale scambia (s_3, \dots, s_7) . Lo scambio $(1, 7)$ sostituisce (s_1, s_3) e (s_7, s_8) con (s_1, s_7) e (s_3, s_8) .

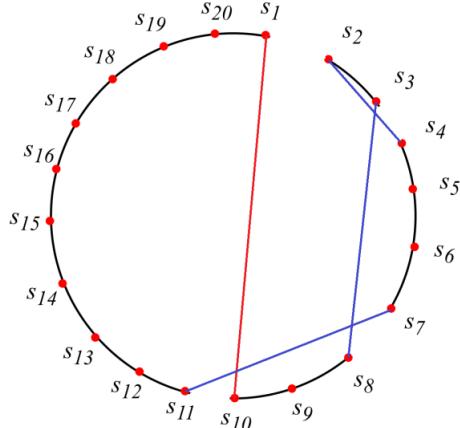


$$x'' = (s_1 \overline{s_7} s_6 s_5 s_4 s_2 s_3 s_8 s_9 s_{10} \overline{s_{11}} s_{12} s_{13} s_{14} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20})$$

Cerca lo scambio migliore che rimuove (s_1, s_7) ed un arco da $x \cap x''$. Assumiamo che questo sia lo scambio $(1, 10)$, il quale sostituisce (s_7, \dots, s_{10}) .

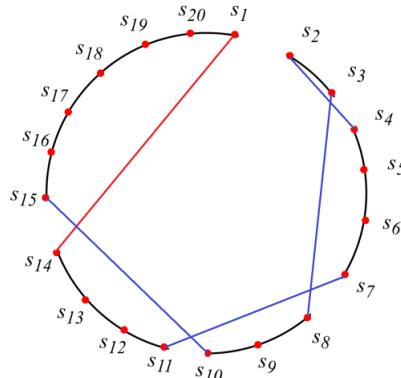
Lo scambio $(1, 10)$ sostituisce (s_1, s_7) e (s_{10}, s_{11}) con (s_1, s_{10}) e (s_7, s_{11}) .

10 RICERCA DI VICINATO SU LARGA SCALA



$$x''' = (s_1 \overline{s_{10} s_9 s_8 s_3 s_2 s_4 s_5 s_6 s_7 s_{11} s_{12} s_{13} s_{14}} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20})$$

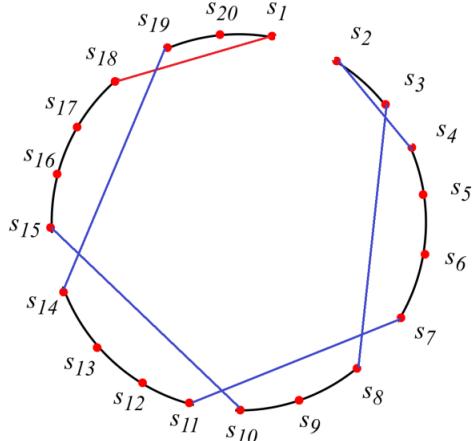
Cerca lo scambio migliore che rimuove (s_1, s_{10}) ed un arco da $x \cap x''$. Assumiamo che questo sia lo scambio $(1, 14)$, il quale sostituisce (s_{10}, \dots, s_{14}) . Lo scambio $(1, 14)$ sostituisce (s_1, s_{10}) e (s_{14}, s_{15}) con (s_1, s_{14}) e (s_{10}, s_{15}) .



$$x^{iv} = (s_1 \overline{s_{14} s_{13} s_{12} s_{11} s_7 s_6 s_5 s_4 s_2 s_3 s_8 s_9 s_{10} s_{15} s_{16} s_{17} s_{18}} s_{19} s_{20})$$

Cerca lo scambio migliore che rimuove (s_1, s_{14}) ed un arco da $x \cap x^{iv}$. Assumiamo che sia $(1, 18)$, il quale sostituisce (s_{14}, \dots, s_{18}) .

Lo scambio $(1, 18)$ sostituisce (s_1, s_{14}) e (s_{18}, s_{19}) con (s_1, s_{18}) e (s_{14}, s_{19}) .



Cercare per il migliore scambio che rimuove (s_1, s_{18}) ed un arco da $x \cap x^v$. Assumiamo che tutti gli scambi restituiscono soluzioni peggiori di x : termina restituendo la migliore soluzione trovata.

10.1.10 Dettagli implementativi

- Ogni passo rimuove un arco dalla soluzione di partenza per evitare di tornare indietro e uno degli archi aggiunti nel passaggio precedente per ridurre la complessità.
- Questo impone un limite superiore sulla lunghezza della sequenza.
- Interrompere la sequenza non appena la soluzione non è più migliore della soluzione iniziale non pregiudica il risultato.

- la variazione totale dell'obiettivo somma gli effetti delle singole mosse

$$\delta f_{o_1, \dots, o_k}(x) = \sum_{\ell=1}^k \delta f_{o_\ell}(x)$$

- ogni sequenza di numeri con somma negativa ammette una permutazione ciclica le cui somme parziali sono tutte negative.
- si ha quindi una permutazione ciclica delle mosse o_1, \dots, o_k con $f < 0$ ad ogni passo.

$$\delta f_{o_1, \dots, o_k}(x) < 0 \implies \exists h : \sum_{\ell=1}^{\ell} \delta f_{o_{h+1}, \dots, o_{h+\ell}}(x) < 0$$

10.1.11 Metodi iterati golosi (destroy-and-repair)

Tutti gli scambi possono essere visti come una combinazione di addizione e rimozione.

$$x' = x \cup A \setminus D$$

Tuttavia:

- gli scambi singoli $x' = x \cup \{j\} \setminus \{i\}$ può dare brutti o risultati infattibili.
- vicinati più grandi possono essere inefficienti.
- in molti problemi le giuste cardinalità di A e D sono sconosciute, perché le soluzioni hanno una cardinalità non-uniforme (e.g, KP, SCP, ...).

Una possibile idea è quella di: rimuovere da x un sottoinsieme $D \subset x$ di cardinalità $\leq k$ e completarlo con una euristica costruttiva. Oppure, aggiungere ad x un insieme $A \subset B \setminus x$ di cardinalità $\leq k$ e ridurlo con un'euristica distruttiva.

Selezione di A e D

Anche se uno dei due sottoinsiemi è costruito con un'euristica polinomiale, di solito non è possibile considerare tutte le possibili scelte per l'altro sottoinsieme.

- il numero dei possibili sottoinsiemi da aggiungere è $O(n^{|A|})$.
- il numero dei possibili sottoinsiemi da rimuovere è $O(n^{|D|})$.

Pertanto la complessità è ulteriormente ridotta:

- generando A (o D) con euristiche randomizzate, distorte per selezionare elementi promettenti in base al loro costo.
- applicando la strategia *fist-best*.

Sono metaeuristiche, anziché euristiche.

10.2 Multi-start, ILS, e VNS

10.2.1 Superare gli ottimi locali

L'euristica di scambio *steepest descent* fornisce solamente l'ottimo locale. Per fare in modo di migliorarla, si può:

- ripetere la ricerca (*come evito di ripercorrere lo stesso cammino?*).
- estendere la ricerca (*come evito di cadere nello stesso ottimo locale?*).

Negli algoritmi costruttivi era possibile solo la ripetizione. La metaeuristica costruttiva sfrutta:

- randomizzazione.

10 RICERCA DI VICINATO SU LARGA SCALA

- memoria.

per operare su $\Delta_A^+(x)$ e $\varphi_A(i, x)$. Le metaeuristiche di scambio li sfruttano per operare su:

1. la soluzione di partenza $x^{(0)}$ (multi-start, ILS, VNS).
2. il vicinato $N(x)$ (VND).
3. il criterio di selezione $\varphi(x, A, D)$ (DLS o GLS, SA, TS).

10.2.2 Condizione di termine

Una ricerca che ripete o procede oltre all'ottimo locale può essere idealmente infinita.

In pratica, uno può utilizzare condizione di terminazione "assolute":

1. un dato numero totale di esplorazioni del vicinato o un dato numero totale di ripetizioni della ricerca locale.
2. un dato tempo totale di esecuzione.
3. un dato valore della funzione obiettivo.
4. un dato miglioramento dell'obiettivo rispetto la soluzione di partenza.

oppure, condizione di terminazione "relative":

1. un dato numero di esplorazioni del vicinato o ripetizioni dopo l'ultimo miglioramento di f^* .
2. un dato tempo di esecuzione dopo l'ultimo miglioramento.
3. un dato valore minimo del rateo tra miglioramento dell'obiettivo e numero di esplorazioni o tempo di esecuzione (es., f^* migliora meno del 1% nelle ultime 1000 esplorazioni).

I confronti equi richiedono condizioni assolute (tempo o numero di esplorazioni).

10.2.3 Modificare la soluzione di partenza: generazione casuale

Risulta possibile creare differenti soluzioni di partenza:

- generandole casualmente.
- applicando differenti euristiche costruttive.
- modificando soluzioni generate da gli algoritmi di scambio.

I vantaggi di una generazione casuale sono:

10 RICERCA DI VICINATO SU LARGA SCALA

- semplicità concettuale.
- rapidità per le problematiche nelle quali è facile garantire la fattibilità.
- controllo sulla distribuzione di probabilità in X in base a:
 - **costo degli elementi** (es., favorire gli elementi più economici).
 - **frequenza degli elementi** durante le ricerche passate, per favorire gli elementi più frequenti (intensificazione) o quelli meno frequenti (diversificazione).
- convergenza asintotica all'ottimo (in tempo infinito).

Gli svantaggi della generazione casuale sono:

- scarsa qualità delle soluzioni di partenza (non quelle finali!).
- lunghi tempi prima di raggiungere l'ottimo locale. Questo dipende dalla complessità degli algoritmi di scambio.
- inefficienza nel decidere la fattibilità è \mathcal{NP} – completa.

10.2.4 Modificando la soluzione di partenza: procedure costruttive

I metodi multi-start sono l'approccio classico:

- progettare molte euristiche costruttive.
- ogni euristica costruttiva genera una soluzione di partenza.
- ogni soluzione di partenza è migliorata da un'euristica di scambio.

Gli svantaggi sono:

1. scarso controllo: le soluzioni generate tendono ad essere simili.
2. impossibilità nel procedere indefinitamente: il numero di ripetizioni è fisso.
3. alto sforzo di progettazione: molti algoritmi diversi devono essere progettati.
4. non garantisce la convergenza, neanche su un tempo infinito.

Conseguentemente, le metaeuristiche costruttive sono attualmente preferite. GRASP e Ant System includono per definizione una procedura di scambio.

10.2.5 Influenza di una soluzione di partenza

Se l'euristica di scambio è:

- buona, la soluzione di partenza ha un'influenza di breve durata: una generazione casuale o euristica di $x^{(0)}$ sono molto simili.
- cattiva, la soluzione di partenza ha un'influenza di lunga durata: una buona euristica per generare $x^{(0)}$ è utile.

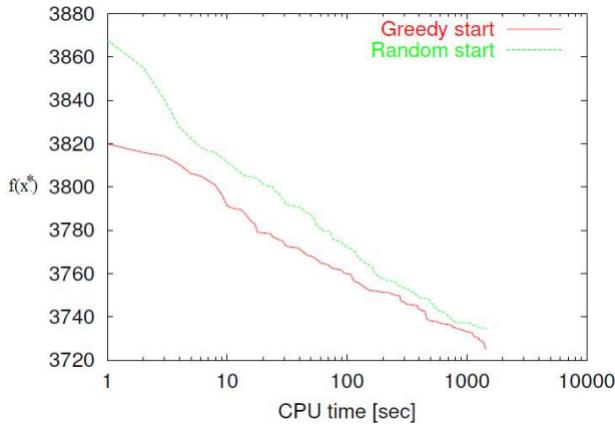


Figura 10.1: L'euristica di scambio non è molto buona.

Modificando la soluzione di partenza sfruttando le precedenti

L'idea è quella di sfruttare le informazioni su le soluzioni precedentemente visitate:

- salvare soluzioni di riferimento, come il miglior ottimo locale trovato finora e possibilmente altri ottimi locali.
- generare la nuova soluzione di partenza modificando quelle di riferimento.

I vantaggi di questo approccio sono:

- controllo: le modifiche possono essere ridotte o incrementate *ad libitum*.
- buona qualità: la soluzione di partenza è molto buona.
- semplicità concettuale.
- semplicità implementativa: la modifica può essere eseguita con le operazioni che definiscono il vicinato.
- convergenza asintotica verso l'ottimo sotto determinate condizioni (adatte).

10.2.6 Iterated Local Search (ILS)

L'algoritmo *Iterated Local Search* (ILS) richiede:

- un'euristica di scambio *steepest descent* per produrre un ottimo locale.
- una procedura di perturbazione per generare soluzioni di partenza.
- una condizione di accettazione per decidere se cambiare il riferimento alla soluzione x .
- una condizione di terminazione.

Algorithm 16 Algoritmo *IteratedLocalSearch*($I, x^{(0)}$)

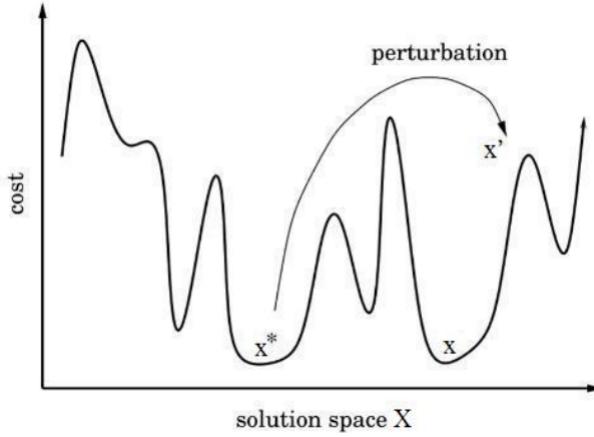
```

 $x := \text{SteepestDescent}(x^{(0)})$ ;
 $x^* := x$ ;
for  $I := 1$  to  $\ell$  do
     $x' := \text{Perturbate}(x)$ ;
     $x' := \text{SteepestDescent}(x')$ ;
    if  $\text{Accept}(x', x^*)$  then
         $x := x'$ ;
    end if
    if  $f(x') < f(x^*)$  then
         $x^* := x'$ ;
    end if
end for
Return  $(x^*, f(x^*))$ ;

```

L'idea è che:

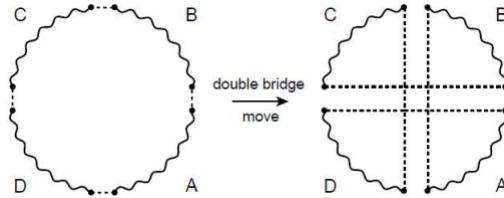
- l'euristica dello scambio esplora rapidamente un bacino di attrazione, terminando in un ottimo locale.
- la procedura di perturbazione si muove in un altro bacino di attrazione.
- la condizione di accettazione valuta se il nuovo ottimo locale è un punto di partenza promettente per la perturbazione successiva.



Esempio: ILS per il TSP

Un applicazione classica di ILS nel TSP utilizza:

- euristiche di scambio: *steepest descent* con vicinato $N_{\mathcal{R}_2}$ o $N_{\mathcal{R}_3}$.
- procedure di perturbazione: un *doppo ponte* muove quel particolare gene-re di 4-scambio.



- condizione di accettazione: la migliore soluzione conosciuta migliora

$$f(x') < f(x^*)$$

10.2.7 Procedura di perturbazione

Sia \mathcal{O} l'insieme delle operazioni che definisce il vicinato $N_{\mathcal{O}}$. La **procedura di perturbazione** esegue un'operazione casuale o , con $o \in \mathcal{O}' \not\subseteq \mathcal{O}$, per evitare che l'euristica di scambio guidi la soluzione x' all'ottimo locale di partenza x .

Due definizioni tipiche di \mathcal{O}' sono:

- sequenze di $k > 1$ operazioni di \mathcal{O} (generare una sequenza casuale è economico).
- concettualmente due operazioni differenti (es., lo scambio di vertici anziché lo scambio di archi).

10 RICERCA DI VICINATO SU LARGA SCALA

La principale difficoltà di ILS risiede nel regolare la perturbazione, nel caso in cui sia:

- troppo forte, trasforma la ricerca in un riavvio casuale.
- troppo debole, riporta la ricerca all'ottimo di partenza.
 - sprecando tempo.
 - possibilmente perdendo la convergenza asintotica.

Idealmente si vorrebbe essere in grado di entrare in un qualsiasi bacino ed uscire da un qualsiasi bacino.

10.2.8 Condizione di accettazione

Algorithm 17 Algoritmo $\text{IteratedLocalSearch}(I, x^{(0)})$

```

 $x := \text{SteepestDescent}(x^{(0)});$ 
 $x^* := x;$ 
for  $I := 1$  to  $\ell$  do
     $x' := \text{Perturbate}(x);$ 
     $x' := \text{SteepestDescent}(x');$ 
    if  $\text{Accept}(x', x^*)$  then
         $x := x';$ 
    end if
    if  $f(x') < f(x^*)$  then
         $x^* := x';$ 
    end if
end for
Return  $(x^*, f(x^*))$ ;

```

La condizione di accettazione equilibra l'intensificazione e la diversificazione.

- L'accettazione delle sole soluzioni migliorative favorisce l'*intensificazione*.

$$\text{Accept}(x', x^*) := (f(x') < f(x^*))$$

La soluzione di riferimento è sempre la migliore trovata: $x = x^*$

- L'accettazione di una qualsiasi soluzione favorisce la *diversificazione*.

$$\text{Accept}(x', x^*) := \text{true}$$

La soluzione di riferimento è sempre l'ultima soluzione ottima trovata: $x = x'$.

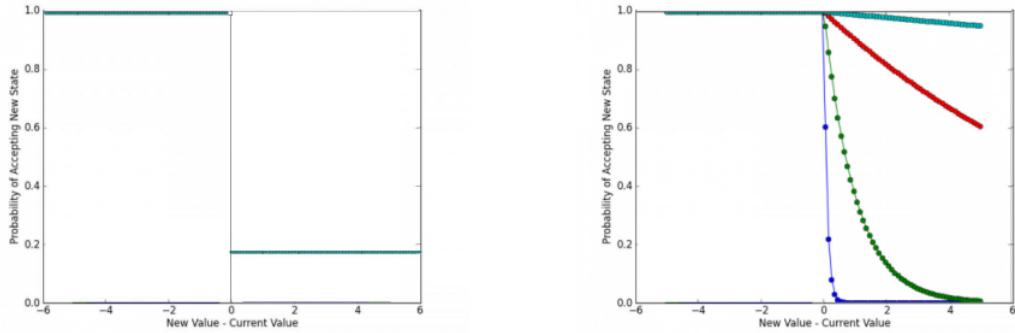
Strategie intermedie possono essere definite basandosi su $\delta f = f(x') - f(x^*)$:

10 RICERCA DI VICINATO SU LARGA SCALA

- se $\delta f < 0$, accetta sempre x' .
- se $\delta f \geq 0$, accetta x' con una probabilità $\pi(\delta f)$, dove $\pi(\cdot)$ è una funzione non crescente.

I casi più tipici sono:

- probabilità costante: $\pi(\delta f) = \bar{\pi} \in (0; 1)$ per ogni $\delta f \geq 0$.
- probabilità monotonamente decrescente con $\pi(0) = 1$ e $\lim_{\delta f \rightarrow +\infty} \pi(\delta) = 0$



Si può usare anche la memoria, accettando x' più facilmente se sono trascorse molte iterazioni dall'ultimo miglioramento di x^* .

10.3 Variable Neighbourhood Search (VNS)

Un metodo molto simile al ILS è il *Variable Neighbourhood Search* (VNS), proposto da Hansen e Mladenović (1997). La principale differenza tra ILS e VNS sono l'utilizzo di:

- la stretta condizione di accettazione: $f(x') < f(x^*)$
- un meccanismo di perturbazione adattivo anziché uno fisso.

VNS introduce spesso anche modifiche di quartiere. Il meccanismo perturbativo si basa su una gerarchia di quartieri, cioè una famiglia di quartieri con dimensione parametrica k crescente.

$$N_1 \subset N_2 \subset \cdots \subset N_k \subset \cdots \subset N_{k_{max}}$$

Tipicamente si utilizzano i vicinati parametrizzati:

- N_{H_k} , in base alla distanza di Hamming tra i sottoinsiemi.
- $N_{\mathcal{O}_k}$, basato sulle sequenze di operazioni da un insieme base \mathcal{O} .

ed estrae $x^{(0)}$ casualmente da un vicinato della gerarchia.

10.3.1 Meccanismo di perturbazione adattivo

Viene chiamato *vicinato variabile* perché il vicinato utilizzato per estrarre $x^{(0)}$ varia in base ai risultati dell'euristica di scambio.

- se viene trovata una soluzione migliore, si utilizza il vicinato più piccolo, per generare una soluzione di partenza molto vicina a x^* (intensificazione).
- se una soluzione peggiore è trovata, si utilizzano vicinati leggermente più grandi, per generare una soluzione di partenza leggermente più distante da x^* (diversificazione).

Questo metodo ha tre parametri:

1. k_{min} identifica il vicinato più piccolo per generare nuove soluzioni.
2. k_{max} identifica il vicinato più grande per generare nuove soluzioni.
3. δk è l'incremento di k tra due tentativi successivi.

L'euristica di scambio adotta il vicinato più piccolo come quello efficiente (N_1 , o comunque N_k con $k \leq k_{min}$).

10.3.2 Schema generale

Algorithm 18 Algoritmo *VariableNeighbourhoodSearch*($I, x^{(0)}$)

```

 $x := SteepestDescent(x^{(0)});$ 
 $x^* := x;$ 
 $k := k_{min};$ 
for  $I := 1$  to  $\ell$  do
     $x' := Shaking(x^*, k);$ 
     $x' := SteepestDescent(x');$ 
    if  $f(x') < f(x^*)$  then
         $x := x';$ 
         $k := k_{min};$ 
    else
         $k := k + \delta k;$ 
    end if
    if  $then k > k_{max}$ 
         $k := k_{min};$ 
    end if
end for
Return  $(x^*, f(x^*))$ ;

```

- la soluzione di riferimento x' è sempre la migliore soluzione conosciuta x^* .

10 RICERCA DI VICINATO SU LARGA SCALA

- la soluzione di partenza si ottiene estraendola a caso dal vicinato corrente della soluzione di riferimento $N_k(x^*)$.
- l'euristica dello scambio produce un ottimo locale rispetto al vicinato di base N .
- se la soluzione più nota migliore, il vicinato attuale diventa $N_{k_{min}}$.
- in caso contrario, spostarsi in un vicinato più grande $N_{k+\delta k}$, senza mai superare $N_{k_{max}}$.

10.3.3 Regolazione dei parametri

Il valore di k_{min} deve essere:

- abbastanza grande da uscire dall'attuale bacino di attrazione.
- abbastanza piccolo da evitare di saltare sui bacini di attrazione adiacenti.

In generale, si pone $k_{min} = 1$, e lo si aumenta nel caso in cui sia sperimentalmente redditizio. Il valore di k_{max} deve essere:

- grande abbastanza da raggiungere una qualsiasi utile bacino di attrazione.
- abbastanza piccolo da evitare di raggiungere zone inutili dello spazio delle soluzioni.

Esempio

Il diametro dello spazio di ricerca per il vicinato di base: $\min(m, n - m)$ per il MDP; n per il TSP e MAX-SAT, ... Il valore di δk deve essere:

- grande abbastanza da raggiungere k_{max} in un tempo ragionevole.
- abbastanza piccolo da consentire ogni ragionevole valore di k .

Generalmente, si pone $\delta k = 1$.

10.3.4 VNS Distorto (skewed)

In modo da favorire la diversificazione. è possibile accettare x' quando

$$f(x') < f(x^*) + \alpha d_H(x', x^*)$$

dove:

- $d_H(x', x^*)$ è la distanza di Hamming tra x' e x^* .
- $\alpha > 0$ è un parametro adatto.

Questo permette di accettare soluzioni peggiorative purché lontane:

- $\alpha \approx 0$ tende ad accettare solamente soluzioni migliorative.
- $\alpha \gg 0$ tende ad accettare qualsiasi soluzione.

Certamente, le strategie casuali viste per l'ILS possono essere adottate.

10.4 VND e DLS

10.4.1 Estendere la ricerca locale senza peggiorarla

Anziché ripetere la ricerca locale la si estende oltre all'ottimo locale. Per evitare il peggiorare delle soluzioni, il passo di selezione deve essere modificato.

$$\tilde{x} := \arg \min_{x' \in N(x)} f(x')$$

Le due strategie principali che permettono questo:

- La **VND** (*Variable Neighbourhood Descent*) cambia il vicinato N .
 - garantisce un'evoluzione senza cicli (l'obiettivo migliora).
 - termina quando tutti i vicinati sono stati sfruttati.
- La **DLS** (*Dynamic Local Search*) cambia la funzione obiettivo f (\tilde{x} è migliore di x per il nuovo obiettivo, forse peggio per il vecchio).
 - può essere intrappolato in cicli (il nuovo obiettivo cambia nel tempo).
 - può procedere indefinitamente.

10.4.2 VND

Il VND di Hansen e Mladenović (1997) sfrutta il fatto che una soluzione è localmente ottima per uno specifico vicinato. Un ottimo locale può essere migliorato utilizzando un vicinato differente. Data una famiglia di vicinati $N_1, \dots, N_{k_{max}}$

1. poni $k := 1$
2. applica un'euristica di scambio *steepest descent* e trova un ottimo locale \bar{x} rispetto a N_k .
3. contrassegna tutti i vicinati per cui \bar{x} è localmente ottima ed aggiorna k .
4. se \bar{x} è un ottimo locale per tutti N_k , termina; altrimenti, torna al punto 2.

Algorithm 19 *VariableNeighbourhoodDescent*($I, x^{(0)}$)

```

 $flag_k := false \forall k;$ 
 $\bar{x} := x^{(0)};$ 
 $x^* := x^{(0)};$ 
 $k := 1;$ 
while  $\exists k : flag_k = false$  do
     $\bar{x} := SteepestDescent(\bar{x}, k);$ 
    if  $f(x') < f(x^*)$  then
         $x^* := \bar{x};$ 
         $flag_{k'} := false \forall k' \neq k;$ 
    else
         $flag_k := true;$ 
    end if
     $k := Update(k);$ 
end while
Return  $(x^*, f(x^*))$ ;

```

Esiste una relazione stretta tra VND e VNS (in fatti furono proposti nello stesso paper). Le differenze fondamentali sono che nel VND:

- ad ogni passo la soluzione corrente è la migliore conosciuta.
- i vicinati sono esplorati, anziché essere utilizzati per estrarre soluzioni casuali (non sono mai grandi).
- i vicinati non formano necessariamente una gerarchia. L'aggiornamento di k non è sempre un incremento.
- quando un ottimo locale per ogni N_k è stato raggiunto, termina. VND è deterministico e troverebbe nient'altro.

Sono presenti due classi principali di metodi VND:

- metodi con *vicinati eterogenei*:
 - sfruttare il potenziale di quartieri topologicamente diversi (es., scambio di vertici anziché di archi).

Conseguentemente, k periodicamente scansiona i valori da 1 a k_{max} (possibilmente effettuando una permutazione casuale della sequenza ad ogni ripetizione).

- metodi con *vicinati gerarchici* ($N_1 \subset \dots \subset N_{k_{max}}$):
 - sfruttare appieno i quartieri piccoli e veloci.
 - ricorrere a quelli grandi e lenti solo per uscire da ottimi locali (solitamente terminando *SteepestDescent* prematuramente).

10 RICERCA DI VICINATO SU LARGA SCALA

Conseguentemente, l'aggiornamento di k funziona come nel VNS:

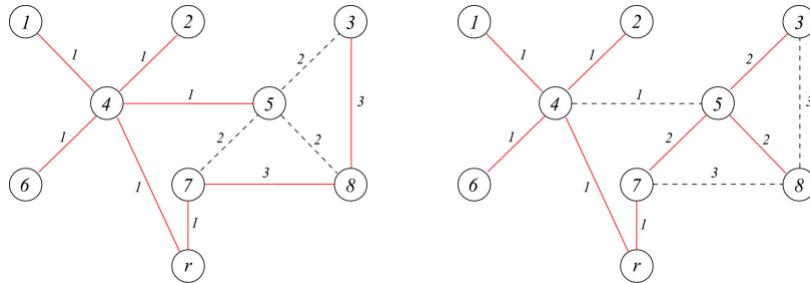
- quando non si trovano miglioramenti in N_k , si incrementa k .
- quando si trovano miglioramenti in N_k , si decrementa k ad 1.

Si termina quando la soluzione corrente è un ottimo locale per tutti i N_k .

- nel caso eterogeneo, si termina quando tutte falliscono.
- nel caso gerarchico, si termina quando la più grande fallisce.

Esempio: il CMSTP

L'istanza del CMSTP ha $n = 9$ vertici, pesi uniformi ($w_v = 1$), capacità $W = 5$ e i costi riportati (gli archi mancanti hanno $c_e \gg 3$)



Considerando il vicinato N_{S_1} (scambi sul singolo arco) per la prima soluzione:

- nessuno arco del ramo destro può essere eliminato, perché il ramo sinistro ha una capacità residua nulla, una connessione diretta alla radice ne aumenterebbe il costo.
- l'eliminazione di qualsiasi arco nel ramo sinistro aumenta il costo totale: la soluzione è un ottimo locale per N_{S_1} .

Il vicinato N_{T_1} (trasferimenti dei vertici singoli) hanno una soluzione migliore, ottenuta muovendo il vertice 5 dal ramo sinistro su quello destro.

10.4.3 DLS

La DLS (*Dynamic Local Search*), conosciuta anche con il nome *Guided Local Search*. Questo è un approccio complementare al VND:

- mantiene il vicinato di partenza.
- modifica la funzione obiettivo.

Viene spesso utilizzata quando l'obiettivo è inutile perché ha ampi *plateau*. L'idea di base è:

- definire una funzione di penalità $w : X \rightarrow \mathbb{N}$
- costruire una funzione ausiliaria $\tilde{f}(f(x), w(x))$
- applicare l'euristica di scambio *steepest descent* per ottimizzare \tilde{f}
- ad ogni iterazione aggiornare la penalità w basata sul risultato

La penalità è adattiva in maniera da allontanarsi dai recenti ottimi locali, ma questo introduce il rischio di finire in un ciclo.

Algorithm 20 *DyamicLocalSearch*($I, x^{(0)}$)

```

 $w := StartingPenalty(I);$ 
 $\bar{x} := x^{(0)};$ 
 $x^* := x^{(0)};$ 
while  $Stop() = false$  do
     $(\bar{x}, x_f) := SteepestDescent(\bar{x}, f, w);$ 
    if  $f(x_f) < f(x^*)$  then
         $x^* := x_f;$ 
    end if
     $w := UpdatePenalty(w, \bar{x}, x^*);$ 
end while
Return  $(x^*, f(x^*))$ ;

```

Notare che l'euristica *steepest descent*:

- ottimizza una combinazione \tilde{f} di f e w .
- restituisce due soluzioni:
 1. una soluzione finale \bar{x} , localmente ottima rispetto a \tilde{f} , per aggiornare w .
 2. una soluzione x_f , che è la migliore conosciuta rispetto ad f .

Varianti

La penalità può essere applicata (per esempio):

- additivamente a gli elementi della soluzione:

$$\tilde{f}(x) = f(x) + \sum_{i \in x} w_i$$

- moltiplicativamente ai componenti della funzione obiettivo $f(x) = \sum_j \phi_j(x)$:

$$\tilde{f}(x) = \sum_j w_j \phi_j(x)$$

10 RICERCA DI VICINATO SU LARGA SCALA

La penalità può essere aggiornata:

- ad ogni singola esplorazione del vicinato.
- quando un ottimo locale per \tilde{f} viene raggiunto.
- quando la migliore soluzione conosciuta x^* rimane invariata per un lungo tempo.

La penalità può essere modificata con:

- **aggiornamenti casuali**: perturbazione "rumorosa" (noisy) dei costi.
- **aggiornamenti basati sulla memoria** (memory-based): favorendo gli elementi più frequenti (intensificazione) o quelli meno frequenti (diversificazione).

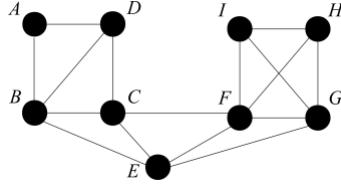
Esempio: DLS per il MCP

Dato un grafo non orientato, trova una cricca di cardinalità massima:

- l'euristica di scambio è un VND utilizzando i vicinati:
 1. N_{A_1} (aggiunta di un vertice): la soluzione migliora sempre, ma il vicinato è molto piccolo e spesso vuoto.
 2. N_{S_1} (scambio di un vertice interno con uno esterno): il vicinato è grande, ma forma un *plateau* (obiettivo uniforme).
- l'obiettivo non fornisce una direzione utile in nessuno dei due vicinati.
- associare ad ogni vertice i una penalità w_i inizialmente uguale a zero.
- l'euristica di scambio minimizza la penalità totale (nel vicinato).
- aggiornare la penalità:
 - quando l'esplorazione di N_{S_1} termina: la penalità della cricca corrente di vertici incrementa di 1.
 - dopo un dato numero di esplorazioni: tutte le penalità non zero decrementano di 1.

La logica del metodo consiste nel mirare a:

- espellere i vertici interni (diversificazione).
- in particolare, i vertici interni più vecchi (memoria).



Si inizia da $x^{(0)} = \{B, C, D\}$, con $w = [0 1 1 1 0 0 0 0 0]$

1. $w(\{B, C, E\}) = w(\{A, B, D\}) = 2$, ma $\{A, B, D\}$ vince lessicograficamente: $x^{(1)} = \{A, B, D\}$ con $w = [1 2 1 2 0 0 0 0 0]$
2. $x^{(2)} = \{B, C, D\}$ con $w = [1 3 2 3 0 0 0 0 0]$ è l'unico vicinato.
3. $w(\{B, C, E\}) = 5 < 7 = w(\{A, B, D\})$: $x^{(3)} = \{B, C, E\}$ con $w = [1 4 3 3 1 0 0 0 0]$
4. $w(\{C, E, F\}) = 4 < 10 = w(\{B, C, D\})$: $x^{(4)} = \{C, E, F\}$ con $w = [1 4 4 3 2 1 0 0 0]$
5. $w(\{E, F, G\}) = 3 < 11 = w(\{B, C, E\})$: $x^{(5)} = \{E, F, G\}$ con $w = [1 4 4 3 3 2 1 0 0]$
6. $w(\{F, G, H\}) = w(\{F, G, I\}) = 3 < 9 = w(\{C, E, F\})$: $x^{(6)} = \{F, G, H\}$ con $w = [1 4 4 3 3 3 2 1 0]$

Ora il vicinato N_{A_1} non è vuoto: $x^{(7)} = \{F, G, H, I\}$

Esempio: DLS per il MAX-SAT

Date m disgiunzioni logiche che dipendono da n variabili logiche, trovare una assegnamento di verità che soddisfi il massimo numero di formule.

- vicinato N_{F_1} (1-flip) è generato complementando una variabile.
- associare ad ogni formula logica una penalità w_j inizialmente uguale ad 1.
- l'euristica di scambio massimizza il peso delle formule soddisfatte modificandone il numero con la penalità moltiplicativa.
- la penalità è aggiornata
 1. incrementando il peso delle formule non soddisfatte per favorirle

$$w_j := \alpha_{us} w_j \text{ per ogni } j \in U(x) \text{ (con } \alpha_{us} > 1)$$

quando un ottimo locale è raggiunto.

11 METAEURISTICHE DI RICOMBINAZIONE

2. riducendo la penalità verso 1

$$w_j := (1 - \rho)w_j + \rho \cdot 1 \text{ per ogni } j \in C \text{ (con } \rho \in (0, 1))$$

con una certa probabilità o dopo un certo numero di aggiornamenti.

La logica del metodo mira a:

- soddisfare le correnti formule non soddisfatte (diversificazione).
- in particolare, quelli che sono rimasti insoddisfatti per più tempo e più recentemente (memoria).

I parametri che regolano l'intensificazione e la diversificazione

- piccoli valori di α_{us} e ρ preservano la penalità corrente (intensificazione).
- grandi valori di α_{us} e ρ cancellano la penalità corrente (diversificazione).

11 Metaeuristiche di ricombinazione

11.1 Euristiche di ricombinazione

Le euristiche costruttive e di scambio gestiscono una soluzione alla volta (eccetto per Ant System). L'euristiche di ricombinazione gestiscono diverse soluzioni in parallelo:

- iniziano da un insieme (detto *popolazione*) di soluzioni (individuali) ottenute in qualche maniera.
- ricombinano gli individui generando una nuova popolazione.

Il loro aspetto originale è l'uso di operazioni che lavorano su più soluzioni, ma spesso includono caratteristiche di altri approcci (a volte rinominati). Alcune sono quasi o completamente deterministiche:

- *Scatter Search*
- *Path Relinking*

altre sono fortemente randomizzate (spesso basate su metafore biologiche):

- algoritmi genetici
- algoritmi memetici
- strategie evolutive

Certamente l'efficacia di un metodo non dipende dalla sua metafora.

11.2 Schema generale

L'idea di base:

- buone soluzioni condividono componenti con l'ottimo globale.
- differenti soluzioni possono condividere differenti componenti.
- combinare differenti soluzioni, è possibile fondere componenti ottimi più facilmente che costruirli passo a passo.

Il tipico schema di un'euristica di ricombinazione è il seguente:

- costruire una popolazione iniziale di soluzioni.
- finché un'adatta condizione di terminazione non produce popolazioni successive (dette *generazioni*).
- per ogni generazione:
 - estraie sottoinsiemi di individui (solitamente, uno o due).
 - applica operazioni di scambio ai singoli individui.
 - applica operazioni di ricombinazione ai sottoinsiemi.
 - colleziona gli individui generati dalle operazioni.
 - sceglie quando accettare o non ogni nuovo individuo (ed in quante copie) generando così una nuova popolazione.

11.3 Scatter Search

SS, proposta da Glover (1977), procede nel seguente modo:

1. genera una popolazione iniziale di soluzioni.
2. migliora tutte le popolazioni con una procedura di scambio.
3. costruisce un insieme di riferimento $R = U \cup D$ dove:
 - il sottoinsieme B contiene le migliori soluzioni conosciute.
 - il sottoinsieme D include le soluzioni più "distanti" (da B e l'uno dall'altro). Questo richiede una definizione di distanza (es., distanza di Hamming).
4. per ogni coppia di soluzioni $(x, y) \in B \times (B \cup D)$
 - "ricombina" x e y , generando z
 - migliora z con una procedura di scambio, ottenendo z'
 - se $z' \notin B$ e B contiene una soluzione peggiore, sostituiscila con z' (non vogliamo duplicati nell'insieme di riferimento)

11 METAEURISTICHE DI RICOMBINAZIONE

- se $z' \notin D$ e D include una soluzione più vicina, sostituisce con z'
(non vogliamo duplicati nell'insieme di riferimento)
5. terminare quando R rimane invariato.

La logica è:

- le ricombinazioni in $B \times B$ intensificano la ricerca.
- le ricombinazioni in $B \times D$ diversificano la ricerca.

Algorithm 21 *ScatterSearch(I, P, n_B, n_D)*

```

 $B := \emptyset;$ 
 $D := \emptyset;$ 
Repeat
  for per ogni  $x \in P$  do
     $z := SteepestDescent(I, x);$ 
    if  $f(z) < f(x^*)$  then
       $x^* := z;$ 
    end if
     $y_B := \arg \max_{y \in B} f(y);$ 
     $y_D := \arg \min_{y \in D} d(y, B \cup D \setminus \{y\})$ 
    if  $z \notin B$  e ( $|B| < n_B$  or  $f(z) < f(y_B)$ ) then
       $B$  tiene le  $n_B$  migliori uniche soluzioni
       $B := B \cup \{z\};$ 
       $Stop := false;$ 
      if  $|B| > n_B$  then
         $B := B \setminus \{y_B\};$ 
      end if
    else if  $z \notin D$  and ( $|D| < n_D$  or  $d(z, B \cup D \setminus y_D) > d(y_D, B \cup D \setminus y_D)$ )
    then
       $D$  tiene le  $n_D$  soluzioni più diverse
       $D := D \cup \{z\};$ 
       $Stop := false;$ 
      if  $|D| > n_D$  then
         $D := D \setminus \{y_D\};$ 
      end if
    end if
  end for
   $P := \emptyset;$ 
  for  $(x, y) \in B \times (B \cup D)$  do
     $P := P \cup Recombine(x, y, I);$ 
  end for
  fino a  $Stop = true;$ 
  Return  $(x^*, f(x^*));$ 

```

11.4 Procedura di ricombinazione

La procedura di ricombinazione dipende dal problema. Solitamente, le soluzioni x e y sono manipolate come sottoinsiemi:

1. include in z tutti gli elementi condivisi da x e y :

$$x := x \cap y$$

(entrambe le soluzioni concorrono nel suggerire quegli elementi)

2. aumentare la soluzione z aggiungendo gli elementi da $x \setminus y$ o $y \setminus x$

- scegliere casualmente o con un criterio di selezione goloso.
- alternativamente da ciascuna fonte o liberamente dalle due fonti (questo è simile ad una euristica costruttiva ristretta).
- se necessario, aggiungere elementi esterni da $B \setminus (x \cup y)$
- se un sottoinsieme z è infattibile, applicare un'euristica di scambio per renderlo fattibile (procedura di riparazione).

Esempi

MDP:

- inizia con $z := x \cap y$
- aumenta z con $k - |z|$ casualmente o punti golosi da x e y .
- non è richiesta alcuna procedura di riparazione

Max-SAT:

- inizia con $z := x \cap y$
- aumenta z con $n - |z|$ casualmente o con assegnamenti di verità golosi da x e y
- non è richiesta alcuna procedura di riparazione

KP:

- inizia con $z := x \cap y$
- aumenta z con elementi casuali o golosi da x e y rispettando la capacità
- non è richiesta alcuna procedura di riparazione, ma la soluzione potrebbe essere aumentata

SCP:

- inizia con $z := x \cap y$
- aumenta z con delle colonne casuali o golose da x e y (evitando quelle ridondanti).
- aggiunge delle colonne esterne da $B \setminus (x \cup y)$
- rimuove le colonne ridondanti con una fase distruttiva

11.5 Path Relinking

PR, proposto da Glover (1989), viene generalmente utilizzato come procedura di intensificazione finale piuttosto che metodo stand-alone. Dato un vicinato N ed un'euristica di scambio basata su questo:

- collezionare in un insieme di riferimento R le migliori soluzioni generate da un'euristica ausiliaria (soluzioni elite).
- per ogni coppia di soluzioni x e y in R
 - si costruisce un cammino γ_{xy} da x a y nello spazio di ricerca del vicinato N , applicando a $z^{(0)}=x$ l'euristica di scambio ausiliaria, ma scegliendo ad ogni passo la soluzione più vicina alla destinazione y .

$$z^{(k+1)} := \arg \min_{z \in N(z^{(k)})} d(z, y)$$

dove d è una opportuna funzione metrica sulle soluzioni. In caso di distanza uguale, ottimizzare la funzione obiettivo f .

- trovare la migliore soluzione z_{xy}^* lungo il cammino

$$z_{xy}^* := \arg \min_{k \in \{1, \dots, |\gamma_{xy}| - 1\}} f(z^{(k)})$$

- se $z_{xy}^* \notin R$ ed è migliore di quelli di R , aggiungilo ad R

Algorithm 22 *PathRelinking*(I, P, n_R)

```

Repeat
   $R := \emptyset;$ 
  for per ogni  $x \in P$  do
     $z := SteepestDescent(I, x);$ 
    if  $f(z) < f(x^*)$  then
       $x^* := z;$ 
    end if
     $y_R := \arg \max_{y \in R} f(y);$ 
    if  $z \notin R$  e ( $|R| < n_R$  or  $f(z) < f(y_R)$ ) then
       $R$  tiene le  $n_R$  migliori uniche soluzioni
       $R := R \cup \{z\};$ 
       $Stop := false;$ 
      if  $|R| > n_R$  then
         $R := R \setminus \{y_R\};$ 
      end if
    end if
  end for
   $P := \emptyset;$ 
  for  $x \in R$  e  $y \in R \setminus \{x\}$  do
     $z := x;$ 
     $z^* := x;$ 
    while  $z \neq y$  do
       $Z := \arg \min_{z' \in N(z)} d(z', y);$ 
       $z := \arg \min_{z' \in Z} f(z');$ 
      if  $f(z) < f(z^*)$  then
         $z^* := z;$ 
      end if
    end while
    if  $z^* \notin P$  then
       $P := P \cup \{z^*\}$ 
    end if
  end for
  Finto a  $Stop = true;$ 
  Return  $(x^*, f(x^*));$ 

```

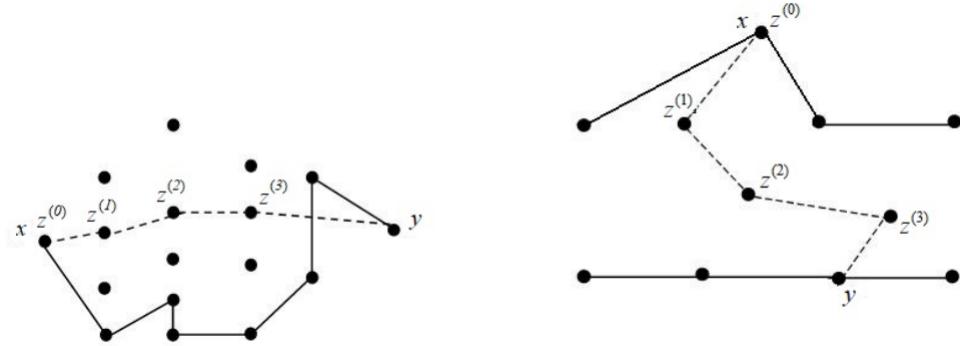
11.5.1 Relinking paths

I cammini esplorati in questa maniera:

- intensificano la ricerca, perché connettono buone soluzioni.
- diversificano la ricerca, perché seguono differenti cammini con rispetto

11 METAEURISTICHE DI RICOMBINAZIONE

alle euristiche di scambio (specialmente se gli estremi sono molto distanti).



- dato che la distanza di $z^{(k)}$ da y sta decrementando, uno può esplorare:
 - soluzioni in peggioramento senza il rischio di comportamenti ciclici
 - sottoinsiemi irrealizzabili senza il rischio di non tornare alla fattibilità (*non migliorano direttamente, ma aprono la strada a miglioramenti*)

Varianti

- *backward path-relinking*: costruisce il cammino al contrario da y a x .
- *back-and-forward path relinking*: costruisce entrambi i cammini.
- *mixed path relinking*: costruisce un cammino con passi alternativi da ciascun estremo.
- *truncated path relinking*: costruisce solo i primi passi del cammino (*se le buone soluzioni sono sperimentalmente vicine l'una con l'altra*).
- *external path relinking*: costruisce un cammino da x muovendosi lontano da y (*se le buone soluzioni sono sperimentalmente lontane l'una con l'altra*).