Università degli Studi di Milano

Algoritmi euristici (laboratorio)

Roberto Cordone

– Anno Accademico 2018/19 –

Indice

Introduzione

Le lezioni di laboratorio del corso di Algoritmi Euristici hanno lo scopo di illustrare gli aspetti pratici del progetto, della realizzazione e della valutazione di algoritmi euristici per problemi di Ottimizzazione Combinatoria.

In queste lezioni si dà per scontata una preparazione di base sulla programmazione in linguaggio C e sugli algoritmi e le strutture dati fondamentali. Le lezioni non si addentreranno quindi nello spiegare tecnicalità su questi punti, ma si limiteranno a richiamare brevemente l'esistenza di istruzioni, algoritmi e strutture che consentono di eseguire le operazioni fondamentali degli algoritmi euristici considerati.

Per semplicità, le lezioni fanno riferimento a un solo problema di Ottimizzazione Combinatoria, cioè il *Maximum Diversity Problem* (in breve, *MDP*). Questo problema è stato scelto fra gli altri trattati nel corso perché la sua definizione è molto semplice e le sue soluzioni sono rappresentabili e manipolabili in modo abbastanza semplice. D'altra parte, il problema è fortemente $\mathcal{NP}$-completo e non ammette garanzie di approssimazione costante, dunque è piuttosto difficile da risolvere all'ottimo. Infine, esso presenta molti aspetti interessanti per quanto riguarda l'efficacia e l'efficienza delle procedure algoritmiche più comuni (inserimenti, scambi, ricombinazioni) e non troppo legati a caratteristiche specifiche del problema.

Il Capitolo 1 definisce il problema e descrive le strutture dati e le procedure di base (realizzate sotto forma di librerie C) che verranno utilizzate per manipolare i dati e le soluzioni. Il Capitolo 2 descrive la realizzazione di alcuni algoritmi costruttivi e distruttivi e la loro valutazione.

Capitolo 1

Il Maximum Diversity Problem

1.1 Definizione

Il *Maximum Diversity Problem* (*MDP*) è definito da:

- un insieme N di *punti* in uno spazio astratto (con $n = |N|$;
- una funzione *distanza* $d : N \times N \to \mathbb{N}$, che associa ad ogni coppia di punti una distanza intera non negativa;
- un numero intero positivo $k \in \mathbb{N}$ con $0 < k < |N|$.

Il problema consiste nel determinare un sottoinsieme $x \subset N$ tale che

- la somma delle distanze dei punti di x sia massima.
- la cardinalità di x sia k;

$$\begin{aligned} \max_{x \subseteq N} f &= \sum_{i \in x} \sum_{j \in x} d_{ij} \\ |x| &= k \end{aligned}$$

Alcune osservazioni secondarie consentono di restringere l'insieme di definizione dei dati, senza ledere la generalità del problema. Per cominciare, nelle applicazioni pratiche le distanze potrebbero essere numeri reali, ma i computer li rappresenteranno sempre con una data precisione, per cui val la pena di considerarli come numeri razionali. D'altra parte, distanze razionali possono essere sempre trasformate in distanze intere cambiando l'unità di misura. Infine, si può sempre ipotizzare che

$$d_{ij} = d_{ji} \text{ per ogni } i, j \in N$$

Infatti, se per ogni coppia di punti (i, j) si sostituiscono d_{ij} e d_{ji} con la loro media aritmetica $(d_{ij} + d_{ji})/2$, il valore di ogni soluzione x rimane lo stesso, dato che la sommatoria $\sum_{i \in x} \sum_{j \in x} d_{ij}$ contiene entrambe le distanze oppure nessuna delle due.

$$d_{ii} = 0 \text{ per ogni } i \in N$$

Infatti, se ogni soluzione contiene k punti, la sommatoria che calcola il valore dell'obiettivo contiene esattamente k distanze d_{ij} per ogni $i \in x$, di cui una è la distanza

d_{ii}; azzerando d_{ii} e sommando $d_{ii}/(k-1)$ ad ogni distanza d_{ij} il valore dell'obiettivo non cambia; la stessa operazione si può fare anche sulle distanze d_{ij} per cui $i \notin x$, dato che non compaiono nell'obiettivo.

$$d_{ij} \geq 0 \text{ per ogni } i, j \in N$$

Infatti, se ogni soluzione contiene k punti, il valore dell'obiettivo è sempre una somma di k^2 termini; se ad ogni distanza si somma un valore costante $\bar{d}$ tale da rendere tutte le distanze non negative, l'obiettivo cresce di $k^2\bar{d}$ per ogni soluzione, e quindi la soluzione ottima non cambia; quindi, val la pena di considerare le distanze come numeri interi non negativi.

Non è invece detto in generale che valga la disuguaglianza triangolare

$$d_{ij} + d_{jk} \geq d_{ik} \text{ per ogni } i, j, k \in N$$

Esistono istanze del *MDP* che ne godono e istanze che non ne godono.

1.2 Rappresentazione dei dati

La libreria `data.h` contiene la struttura

```
typedef struct data_s
 {
  int n;   // cardinalita' dell'insieme N dei punti
  int k;   // cardinalita' delle soluzioni ammissibili x \in X
  int **d; // matrice delle distanze fra punti (simmetrica)
 } data_t;
```

che useremo per rappresentare ciascuna istanza I data del *MDP*, dato che essa consiste solo di tre componenti: un insieme N, una metrica d e un numero k.

Rappresenteremo l'insieme N con i numeri naturali da 1 a $n = |N|$, dato che non ha altre specificazioni oltre alla metrica. Non adottiamo la convenzione del linguaggio C, che rappresenta gli insiemi di numeri partendo da zero, per rimanere coerenti con i file di dati disponibili in letteratura, e per poter usare l'indice 0 in eventuali altre operazioni.

Rappresenteremo la metrica d con una matrice quadrata di distanze. Si potrebbe risparmiare spazio sfruttandone la simmetria, cioè rappresentando solo i valori d_{ij} con $i < j$, ma questo comporterebbe ad ogni accesso al dato d_{ij} un test per valutare se $i < j$ o no, e un eventuale scambio dei due indici. Siccome prevediamo di dover accedere moltissime volte ai dati, scegliamo di rappresentarli tutti per privilegiare l'efficienza temporale su quella spaziale.

La matrice delle distanze è dinamica, e verrà allocata caricando i dati da file e deallocata alla fine dell'elaborazione. È fornita una libreria `alloc.h` che fornisce funzioni per allocare vettori di interi (indicizzati da 1 a n) e matrici di numeri interi (indicizzate da 1 a $n1$ sulle righe e da 1 a $n2$ sulle colonne).

```
// Alloca un vettore di n int
int *int_alloc (int n);

// Alloca una matrice di (n1,n2) int
int **int2_alloc (int n1, int n2);
```

Per gestire i dati, la libreria `data.h` fornisce funzioni per caricare i dati da un file (nel formato standard AMPL), per deallocare la struttura dati `data_t` sopra descritta e per stampare i dati a video (ancora nel formato AMPL).

```
// Carica dal file data_file i dati dell'istanza puntata da *pI
void load_data (char *data_file, data_t *pI);

// Dealloca le strutture dell'istanza puntata da *pI
void destroy_data (data_t *pI);

// Stampa l'istanza puntata da *pI
void print_data (data_t *pI);
```

1.3 Istanze di benchmark

Esistono in letteratura diverse classi di istanze di benchmark per il *MDP*. Nel seguito, per brevità, usiamo un semplice gruppo di 20 istanze, detto *SOM*, dalle iniziali dei nomi dei tre autori che le proposero nel 2003 (G.C. Silva, L.S. Ochi e S.L. Martins). In queste istanze:

- l'insieme N contiene $n = 100$, 200, 300, 400 e 500 punti;
- il numero k è pari a $0.1n$, $0.2n$, $0.3n$, $0.4n$;
- le distanze d_{ij} sono numeri interi casuali distribuiti uniformemente fra 0 e 9;
- le distanze sono simmetriche, non negative e nulle fra un punto e sé stesso.

Non sono istanze particolarmente significative o realistiche, ma sono state ampiamente utilizzate in letteratura per testare algoritmi euristici per l'*MDP*. Sono abbastanza difficili da non essere banalmente risolte all'ottimo da qualsiasi metodo, ma abbastanza facili da far sì che la miglior soluzione nota sia probabilmente ottima (anche se questo non è ancora stato dimostrato) e abbastanza piccole da richiedere tempi ragionevoli ad algoritmi di complessità polinomiale.

I dati sono conservati in file di testo. Il nome del file determina le caratteristiche dell'istanza: il file `[%i]matrizn[%n]m[%k].dat` è l'`%i`-esimo su 20, l'insieme N include `%n` punti e la cardinalità richiesta per la soluzione è `%k`.

Il contenuto del file è in formato AMPL, uno dei formati standard nella rappresentazione di problemi a numeri interi, usato da modellatori e risolutori general-purpose di tali problemi. Il formato (che del resto non ci riguarda, dato che la libreria `data.h` fornisce una procedura per caricare i dati nelle strutture in memoria), è autoesplicativo.

```
param n := 100 ;
param m := 10 ;
param D :=
[1,1] 0 [1,2] 0 [1,3] 3 [1,4] 4 [1,5] 7 ...
[2,1] 0 [2,2] 0 [2,3] 2 [2,4] 5 [2,5] 4 ...
...
```

1.4 Rappresentazione delle soluzioni

Le soluzioni dell'*MDP*, come di tutti i problemi di Ottimizzazione Combinatoria, sono sottoinsiemi dell'insieme base. Ci sono due modi principali di rappresentare un sottoinsieme:

1. con un *vettore di incidenza*, che fa corrispondere ad ogni $i \in N$ un valore booleano
$$x_i = \begin{cases} \texttt{true} \text{ quando } i \in x \\ \texttt{false} \text{ quando } i \in N \setminus x \end{cases}$$
2. con una *lista di elementi*, che consente di scorrere i soli elementi della soluzione $i \in x$

La scelta tra le due rappresentazioni dipende dal tipo di operazioni da compiere negli algoritmi: il test di appartenenza di un punto alla soluzione è efficiente nella prima rappresentazione ($O(1)$), inefficiente nella seconda ($O(n)$); l'opposto vale per lo scorrimento dei soli elementi interni o esterni alla soluzione. Negli algoritmi che ci interessa realizzare vengono usate soprattutto le operazioni su liste, sia quella dei punti interni sia quella dei punti esterni alla soluzione. Per esempio, gli algoritmi costruttivi scorrono ad ogni passo l'insieme $\mathrm{Ext}_A(x)$, che nell'*MDP* coincide con il complemento della soluzione, $N \setminus x$, mentre il calcolo del valore di una soluzione, $f(x)$, comporta di scorrere i suoi elementi. Tuttavia, adotteremo entrambe le rappresentazioni per mantenere flessibilità. A posteriori, potrebbe valere la pena di cancellarne una, se non usata. La libreria `solution.h` contiene la struttura

```
typedef struct solution_s
{
  int f;         // valore della soluzione

  int card_x;    // cardinalita' della soluzione x
  int card_N;    // cardinalita' dell'insieme N

  bool *in_x;    // per ogni punto i indica se appartiene o no a x

  // Liste dei punti nella soluzione x e nel complemento N \setminus x
  int head_x;    // sentinella della lista dei punti in x
  int head_notx; // sentinella della lista dei punti in N \setminus x
  int *next;     // elemento seguente   per ciascun punto
  int *prev;     // elemento precedente per ciascun punto

} solution_t;
```

Il valore f della soluzione viene conservato e aggiornato, in modo da potervi accedere in tempo costante $O(1)$ anziché ricalcolarlo ogni volta. Il ricalcolo richiederebbe tempo $O(n^2)$ se fatto scorrendo il vettore di incidenza, tempo $O(k(n-k))$ se fatto scorrendo le due liste x e $N \setminus x$.

Nota : essendo la matrice delle distanze simmetrica e intera, $f(x)$ è certamente pari, dato che è una somma di coppie di termini uguali. È convenzione comune riportare in `f` la metà della somma di tutte le distanze. Nel seguito ne riparleremo.

La cardinalità della soluzione $|x|$ viene conservata esplicitamente, anche se dovrebbe essere fissata dai dati al valore k, perché in questo modo la stessa struttura può essere usata per rappresentare anche le *soluzioni parziali*, cioè i sottoinsiemi di cardinalità $< k$. Infatti, abbiamo scelto come spazio di ricerca $\mathcal{F}$ proprio l'insieme delle soluzioni parziali. La cardinalità dell'insieme N a rigore non avrebbe posto qui, ma vedremo che è utile per gestire la rappresentazione della soluzione come lista di punti.

Il vettore booleano `in_x` rappresenta il vettore di incidenza. Il tipo `boolean`, con i valori `false` e `true` viene fornito dalla libreria `defs.h`.

```
enum _bool {false = 0, true = 1};
typedef enum _bool bool;
```

e i vettori dinamici di booleani (indicizzati da 1 a n) possono essere allocati grazie alla libreria `alloc.h` già citata, che fornisce la funzione

```
// Alloca un vettore di n bool
bool *bool_alloc (int n);
```

Le due liste che rappresentano la soluzione x e il suo complemento $N \setminus x$ sono *liste doppie circolari con sentinella*, in modo da poter eseguire qualsiasi operazione fondamentale (inserimento, estrazione, ecc...) in tempo costante, al costo di una maggior occupazione di memoria. Brevemente, tali liste possono essere scorse in entrambi i versi, e non sono mai fisicamente vuote, in quanto contengono sempre un elemento fittizio, detto sentinella (per convenzione, si definisce vuota una lista che contiene solo la sentinella).

Adotteremo la rappresentazione delle due liste con vettori e indici interi, anziché quella con puntatori e strutture allocate dinamicamente, perché i punti sono definiti una volta per tutte all'inizio dell'algoritmo, e solo la loro posizione varia dinamicamente durante l'esecuzione. Inoltre, siccome le due liste sono complementari, e quindi non hanno intersezioni, sfrutteremo gli stessi vettori e gli stessi indici `next` e `prev` per le due liste; solo le teste `head_x` e `head_notx` saranno diverse.

Per non perderci in tecnicalità di programmazione, nasconderemo tutti i dettagli implementativi incapsulando l'implementazione così che ogni accesso ai dati avvenga con funzioni di libreria. Questo consente anche di modificare l'implementazione a basso livello senza che gli algoritmi già realizzati ne risentano. Questo può comportare qualche inefficienza temporale, perché richiede chiamate di funzione là dove basterebbe un semplice accesso a strutture dati. Accettiamo questa inefficienza perché si supera facilmente usando macro (o definizioni `inline` in C++). Però non descriviamo il modo di superarla per non perderci in tecnicalità.

La libreria `solution.h` fornisce alcune funzioni per gestire la soluzione nel suo complesso:

```
// Crea una soluzione vuota per un problema di dimensione n
void create_solution (int n, solution_t *px);

// Dealloca la soluzione puntata da *px
void destroy_solution (solution_t *px);

// Indica se la soluzione puntata da *px e' vuota
bool empty_solution (solution_t *px);

// Copia la soluzione puntata da *px_orig nella soluzione puntata da *px_dest
void copy_solution (solution_t *px_orig, solution_t *px_dest);

// Stampa la soluzione puntata da *px
void print_solution (solution_t *px);
```

La creazione di una soluzione vuota corrisponde al tipico primo passo di un'euristica costruttiva, che parte con il sottoinsieme vuoto. La deallocazione viene eseguita al termine dell'algoritmo. La copia è utile quando la soluzione corrente è

la migliore nota sinora, e quindi va salvata a parte. La stampa è utile in fase di analisi dei risultati: riporta a video su una sola riga il nome del file dei dati, il valore dell'obiettivo, la lista dei punti di x.

Per rappresentare un punto, occorre semplicemente conoscerne l'indice, che è un numero intero compreso fra 1 e n. Cerchiamo (forse con un eccesso di zelo) di tener conto di possibili sviluppi futuri nei quali a un punto potrebbero essere associate informazioni ausiliarie (ad es., coordinate, stringhe di testo, ecc...). Per non trovarsi a dover riscrivere tutti gli algoritmi, introduciamo un livello di astrazione che distingua:

- da un lato, l'indice numerico intero di tipo `int`;
- dall'altro il punto astratto, che raccoglie tutte le informazioni associate, di tipo `point`.

Concretamente, i due oggetti sono la stessa cosa, grazie alla definizione di tipo:

```
typedef int point_pos;
```

ma utilizzeremo variabili del primo tipo per scorrere gli indici (per esempio, nel vettore di incidenza o nella matrice dei dati) e variabili del secondo tipo per scorrere i punti veri e propri (per esempio, nella lista dei punti interni x e in quella dei punti esterni $N \setminus x$). Per passare da una all'altra informazione useremo le funzioni:

```
// Legge l'indice del punto p nella soluzione puntata da *px
int get_index (point p, solution_t *px);

// Determina il punto p di indice i nella soluzione puntata da *px
point get_point (int i, solution_t *px);
```

che al momento si limitano a riportare in uscita il valore fornito in ingresso, mentre in caso di modifiche delle strutture dati si occuperanno delle relative conversioni. Questo è l'esempio più chiaro delle inefficienze a cui si è accennato prima. È il prezzo che paghiamo al lusso di poter ignorare l'implementazione fisica della struttura, e di poterla modificare in qualsiasi momento successivo senza toccare gli algoritmi ad alto livello.

Per scorrere la lista x senza fare riferimento alla sua implementazione concreta useremo le apposite funzioni di libreria:

```
// Restituiscono il primo e l'ultimo punto della soluzione *px
point first_point_in (solution_t *px);
point last_point_in (solution_t *px);

// Restituiscono il punto che segue e quello che precede p
point next_point (point p, solution_t *px);
point prev_point (point p, solution_t *px);

// Indica se si e' arrivati in fondo alla lista (p e' una sentinella)
bool end_point_list (point p, solution_t *px);
```

A semplice titolo di curiosità, la lista x usa come sentinella gli elementi di indice 0 dei due vettori: quindi, per sapere se si è arrivati in fondo alla lista, basta verificare se `p` è compreso fra 1 e `card_N`.

Per scorrere la lista complementare $N \setminus x$, cambiano le funzioni di accesso al primo e ultimo punto. dato che la sentinella corrisponde agli elementi di indice `card_N+1` dei due vettori:

```
// Restituiscono i cursori al primo e all'ultimo punto fuori della soluzione puntata da *px
point first_point_out (solution_t *px);
point last_point_out (solution_t *px);
```

Non cambiano invece le funzioni per passare agli elementi successivi o precedenti, dato che le due liste non si intersecano mai e usano gli stessi vettori con le stesse regole. Anche la funzione per determinare se si è in fondo alla lista non cambia, dato che si tratta sempre di verificare se `p` è compreso fra 1 e `card_N`.

Ad esempio, avendo dichiarato i punti `p` e `q`

```
point p, q;
```

per scorrere la soluzione x dal primo all'ultimo elemento, si eseguirà il ciclo

```
for (p = first_point_in(&x); !end_point_list(p,&x); p = next_point(p,&x))
```

e per scorrere il complemento $N \setminus x$ dall'ultimo al primo elemento, si eseguirà il ciclo

```
for (q = last_point_out(&x); !end_point_list(q,&x); q = prev_point(q,&x))
```

Le operazioni principali di manipolazione della soluzione sono l'aggiunta di un singolo punto alla soluzione o la sua eliminazione. Entrambe comportano di spostare un punto da una delle due liste all'altra, dato che esse sono rigorosamente complementari. È bene che quindi esistano delle funzioni specifiche per queste operazioni, mentre non sono necessarie funzioni per il semplice inserimento o la semplice estrazione, che potrebbero solo favorire errori negli algoritmi (queste funzioni potrebbero esserci, ma allora conviene che restino nascoste nella libreria).

```
// Aggiunge il punto di indice i alla soluzione *px
void move_point_in (int i, solution_t *px, data_t *pI);

// Cancella il punto di indice i dalla soluzione *px
void move_point_out (int i, solution_t *px, data_t *pI);
```

Queste funzioni si occupano di mantenere aggiornate e coerenti tutte le componenti della struttura dati. Dato l'indice `i`, la funzione `move_point_in`[1]:

1. aggiunge alla funzione obiettivo le distanze del nuovo punto dai precedenti (ma non viceversa perché in `f` riportiamo solo metà dell'obiettivo); questa operazione richiede di conoscere i dati dell'istanza;
2. aumenta di un'unità la cardinalitá `card_x`;
3. pone uguale a `true` il valore di `in_x` in corrispondenza all'indice `i`;
4. trova il punto `p` di indice `i` nella lista del complemento e lo estrae;
5. aggiunge il punto `p` alla lista della soluzione.

Dato l'indice `i`, la funzione `move_point_out`:

1. toglie dalla funzione obiettivo le distanze del nuovo punto dai precedenti (ma non viceversa perché in `f` riportiamo solo metà dell'obiettivo); questa operazione richiede di conoscere i dati dell'istanza;

[1] Si potrebbe discutere se convenga passare alla funzione il punto `p` anziché l'indice `i`. Nel nostro caso, è ovviamente lo stesso.

2. diminuisce di un'unità la cardinalitá `card_x`;
3. pone uguale a `false` il valore di `in_x` in corrispondenza all'indice `i`;
4. trova il punto `p` di indice `i` nella lista della soluzione e lo estrae;
5. aggiunge il punto `p` alla lista del complemento.

Tutte queste operazioni richiedono tempo costante, tranne l'adeguamento della funzione obiettivo, che richiede tempo $O(|x|)$ per l'aggiunta e tempo $O(n - |x|)$ per la cancellazione.

Si noti che l'uso di rappresentazioni multiple della soluzione in parallelo comporta un carico aggiuntivo di lavoro, dedicato a mantenere aggiornate e coerenti le varie rappresentazioni. Questa scelta deve essere giustificata dall'efficienza acquistata altrove.

Inoltre, questa scelta espone al rischio che, per qualche motivo, la coerenza si perda. Nel nostro caso, la struttura dati `solution_t` contiene cinque campi potenzialmente incoerenti tra loro: obiettivo, cardinalità, vettore di incidenza, lista dei punti in soluzione e lista dei punti nel complemento. Le funzioni fornite dalla libreria dovrebbero garantire che queste cinque sottostrutture siano sempre coerenti. Anche ammettendo che lo siano in partenza, ogni successiva modifica (per esempio dovuta all'aggiunta di altri campi alla soluzione, per poter eseguire altre operazioni o per eseguire più velocemente le stesse operazioni) potrebbe introdurre incoerenze, e quindi errori. Lavorando su algoritmi euristici, è un'ottima norma introdurre una funzione di verifica della coerenza interna delle strutture dati.

```
// Verifica la coerenza interna della soluzione puntata da *px in base
// all'istanza puntata da *pI, partendo dal vettore di incidenza
bool check_solution (solution_t *px, data_t *pI);
```

Questa funzione assume come valida a priori una delle cinque componenti, e ricalcola le altre quattro, verificando se i loro valori correnti sono corretti. La scelta della componente valida è arbitraria, purché essa sia sufficiente a ricavare il valore delle altre. In generale, si usa la più semplice, cioè quella che più difficilmente può risultare sbagliata. Nel nostro caso, si dà per buono il vettore di incidenza, e se ne ricavano obiettivo, cardinalità e liste: se si scoprono incoerenze, la funzione restituisce il valore `false`, e si può decidere di interrompere l'esecuzione per correggere il codice.

1.5 Il programma principale

La funzione `main` gestisce la definizione dei parametri degli algoritmi, il caricamento dei dati, le allocazioni e deallocazioni di dati e soluzione, la scelta degli algoritmi da eseguire, la determinazione del tempo di calcolo e la stampa dei risultati a video.

```
  parse_command_line(argc,argv,data_file,param);

  load_data(data_file,&I);

  create_solution(I.n,&x);

  inizio = clock();
  if (strcmp(param,"-g") == 0)
    greedy(&I,&x);
```

```
else if (strcmp(param,"-gbs") == 0)
  greedy_bestsum(&I,&x);
else if (strcmp(param,"-gbp") == 0)
  greedy_bestpair(&I,&x);
else if (strcmp(param,"-gta") == 0)
  greedy_tryall(&I,&x);
fine = clock();
tempo = (double) (fine - inizio) / CLOCKS_PER_SEC;

printf("%s ",data_file);
printf("%10.6lf ",tempo);
print_solution(&x);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

Per semplicità, si è considerato un solo parametro (`param`), di tipo stringa che useremo per distinguere tutti gli algoritmi descritti nel seguito.

La stampa dei risultati avviene tutta su una riga, in modo da poter raccogliere in uno *script* una serie di chiamate del programma su altri dati o con altri parametri, e accodare via via i loro risultati, uno per riga.

Capitolo 2

Algoritmi costruttivi

2.1 Schema generale

Gli algoritmi costruttivi seguono lo schema generale descritto nelle lezioni di teoria:

Algorithm Greedy(I)
$x := \emptyset$;
$x^* := \emptyset$; $f^* := +\infty$; { Miglior soluzione trovata sinora }
While $\mathrm{Ext}_A(x) \neq \emptyset$ *do*
    $i := \arg \min_{i \in \mathrm{Ext}_A(x)} \varphi_A(i, x)$;
    $x := x \cup \{i\}$;
    If $x \in X$ and $f(x) < f^*$ *then* $x^* := x$; $f^* := f(x)$;
Return (x^*, f^*);

Adattiamo questo schema generale al caso specifico dell'*MDP*. Per prima cosa, la semplicissima struttura delle soluzioni ammissibili (l'unico vincolo è la cardinalità fissata) suggerisce di definire lo spazio di ricerca come insieme delle soluzioni parziali, cioè dei sottoinsiemi con non più di k punti.

$$\mathcal{F}_A = \{x \subseteq N : |x| \leq k\}$$

Ne deriva che l'insieme delle estensioni possibili per una data soluzione parziale coincide con il suo complemento (eccetto nell'ultimo passo, quando l'estensione possibile è vuota)

$$\mathrm{Ext}_A(x) = \begin{cases} N \setminus x & \text{per } |x| < k \\ \emptyset & \text{per } |x| = k \end{cases}$$

e che l'unica soluzione ammissibile determinata dall'algoritmo durante la propria esecuzione è l'ultima. Inoltre, trattandosi di un problema di massimizzazione, è più naturale considerare anche il criterio di scelta come una funzione da massimizzare.

Questo trasforma lo schema generale come segue:

Algorithm GreedyMDP(I)
$x := \emptyset$;
While $|x| < k$ *do*
$i := \arg\max_{i \in N \setminus x} \varphi_A(i, x)$;
$x := x \cup \{i\}$;
Return (x, f);

Realizzare questo schema con le funzioni di libreria è decisamente semplice:

```
void greedy (data_t *pI, solution_t *px)
{
  int i;

  while (px->card_x < pI->k)
  {
    i = best_additional_point(px,pI);
    move_point_in(i,px,pI);
  }
}
```

L'istruzione $x := \emptyset$ corrisponderebbe a una chiamata `create_solution(pI->n,px);`, ma si è preferito eseguirla all'esterno come `create_solution(I.n,&x);` e passare la soluzione vuota così ottenuta come parametro alla funzione `greedy`. Il vantaggio di questo procedimento è che consente di usare la funzione `greedy` non solo per produrre una soluzione da zero, ma anche per completare un'eventuale soluzione parziale ottenuta in qualche altro modo.

La funzione `best_additional_point(px,pI)` va implementata determinando l'indice `i` del punto migliore da aggiungere alla soluzione `*px` in base ai dati dell'istanza `*pI` secondo il criterio di scelta $\varphi_A(i, x)$, che ancora non abbiamo definito. Definizioni diverse daranno luogo a diversi algoritmi costruttivi.

2.2 L'algoritmo costruttivo base

Dato che la funzione obiettivo si può facilmente definire anche su soluzioni parziali, la definizione più semplice per il criterio di scelta è il valore dell'obiettivo, cioè

$$\varphi_A(i, x) = f(x \cup \{i\}) = \sum_{j \in x \cup \{i\}} \sum_{k \in x \cup \{i\}} d_{jk}$$

Valutarla da zero richiede un tempo $O\left(|x|^2\right)$, ma non è effettivamente necessario, dato che è sufficiente massimizzarla. Per ottenere questo risultato, si può considerare invece la variazione $\delta f(x, i) = f(x \cup \{i\}) - f(x)$

$$\delta f(x, i) = f(x \cup \{i\}) = \sum_{j \in x} d_{ji} + \sum_{j \in x} d_{ij} - d_{ii} = 2 \sum_{j \in x} d_{ji}$$

che richiede solo tempo $O(|x|)$ per il calcolo (e ovviamente non occorre moltiplicarla per 2).

L'osservazione precedente permette di realizzare l'istruzione

$$i := \arg\max_{i \in N \setminus x} f(x \cup \{i\});$$

con la semplice chiamata

```
i = best_additional_point(px,pI);
```

della funzione

```
int best_additional_point (solution_t *px, data_t *pI)
{
  point p;
  int d, d_max;
  int i, i_max;

  d_max = -1;
  i_max = NO_POINT;
  for (p = first_point_out(px); !end_point_list(p,px); p = next_point(p,px))
  {
    i = get_index(p,px);
    d = dist_from_solution(i,px,pI);
    if (d > d_max)
    {
      i_max = i;
      d_max = d;
    }
  }

  return i_max;
}
```

che calcola per ogni punto di $N \setminus x$, di indice `i`, la variazione della funzione obiettivo $\delta f(x,i)/2 = \sum_{j \in x} d_{ji}$ ottenuta aggiungendo il punto alla soluzione, cioè la distanza del punto dalla soluzione stessa. Questo valore viene calcolato dalla funzione `dist_from_solution(i,px,pI)`. Di tutti i valori si conserva il massimo, con il corrispondente indice `i_max`. Il risultato è l'algoritmo costruttivo base.

2.2.1 Sperimentazione

Ora procediamo a lanciare l'algoritmo sull'intero benchmark. Lo *script* `greedy_solve.bat` applica l'algoritmo e ne redirige l'uscita dal video sul file `report.txt`.

```
greedy dati\01matrizn100m10.dat >  report.txt
greedy dati\02matrizn100m20.dat >> report.txt
greedy dati\03matrizn100m30.dat >> report.txt
greedy dati\04matrizn100m40.dat >> report.txt
greedy dati\05matrizn200m20.dat >> report.txt
greedy dati\06matrizn200m40.dat >> report.txt
greedy dati\07matrizn200m60.dat >> report.txt
greedy dati\08matrizn200m80.dat >> report.txt
greedy dati\09matrizn300m30.dat >> report.txt
greedy dati\10matrizn300m60.dat >> report.txt
greedy dati\11matrizn300m90.dat >> report.txt
greedy dati\12matrizn300m120.dat >> report.txt
greedy dati\13matrizn400m40.dat >> report.txt
greedy dati\14matrizn400m80.dat >> report.txt
greedy dati\15matrizn400m120.dat >> report.txt
```

```
greedy dati\16matrizn400m160.dat >> report.txt
greedy dati\17matrizn500m50.dat >> report.txt
greedy dati\18matrizn500m100.dat >> report.txt
greedy dati\19matrizn500m150.dat >> report.txt
greedy dati\20matrizn500m200.dat >> report.txt
```

La prima chiamata applica la redirezione in scrittura (`>`), le successive la redirezione in *append* (`>>`), in modo da ottenere un sommario molto regolare, con un'istanza per ogni riga.

```
dati\01matrizn100m10.dat   0.000000 f =    306   59   19  100   26 ...
dati\02matrizn100m20.dat   0.000000 f =   1137   31   51   36   35 ...
dati\03matrizn100m30.dat   0.000000 f =   2343   52   94   28   37 ...
dati\04matrizn100m40.dat   0.000000 f =   4055   36   94   67   30 ...
dati\05matrizn200m20.dat   0.000000 f =   1177  173   16   71  110 ...
dati\06matrizn200m40.dat   0.000000 f =   4333   40  159   64   19 ...
dati\07matrizn200m60.dat   0.016000 f =   9263  124  122  190  181 ...
dati\08matrizn200m80.dat   0.016000 f =  16046   90  138  168    8 ...
dati\09matrizn300m30.dat   0.000000 f =   2628   41  240  178  251 ...
dati\10matrizn300m60.dat   0.000000 f =   9449   81  225   24  256 ...
dati\11matrizn300m90.dat   0.000000 f =  20542  134  182  264  279 ...
dati\12matrizn300m120.dat   0.000000 f =  35592  173  157  228  197 ...
dati\13matrizn400m40.dat   0.016000 f =   4531   96   44  312   25 ...
dati\14matrizn400m80.dat   0.000000 f =  16653  228   20  360  182 ...
dati\15matrizn400m120.dat   0.031000 f =  35937  169  167   66  162 ...
dati\16matrizn400m160.dat   0.031000 f =  62091   13  234  138  235 ...
dati\17matrizn500m50.dat   0.016000 f =   6917  128   55  264  197 ...
dati\18matrizn500m100.dat   0.031000 f =  25605  240   81  288  337 ...
dati\19matrizn500m150.dat   0.047000 f =  55936  366  116  293  205 ...
dati\20matrizn500m200.dat   0.046000 f =  96543  370  431  350  473 ...
```

Ci interessano le colonne con il risultato ($f(x)$) e con il tempo di calcolo. Per valutare la qualità dei risultati, è opportuno calcolare i gap relativi $\delta(x) = |f(x) - f^*| / f^*$, in modo che i valori relativi a istanze diverse siano confrontabili in modo più ragionevole. Poiché non conosciamo gli ottimi f^*, al loro posto si dovrebbe usare una stima per eccesso al numeratore e una per difetto al denominatore, in modo da ottenere complessivamente una stima per ecdesso del gap. Questo vale perché il problema è di massimo: se fosse di minimo, si userebbe in entrambi i casi una stima per difetto. Succede però che per l'*MDP* le stime per eccesso sono di qualità piuttosto scarsa, e comunque molto più lontane dall'ottimo delle stime per difetto fornite dalle migliori soluzioni euristiche note. Quindi, useremo sia al numeratore sia al denominatore i migliori risultati noti in letteratura. Questo implica che le nostre stime del gap potrebbero essere sbagliate in entrambe le direzioni, ma l'errore è probabilmente piccolo.

Analisi dei tempi di calcolo Per condurre analisi sensate, occorrerebbe un benchmark di istanza molto più ricco. Ignoriamo questo aspetto e procediamo con qualche analisi, giusto per esporre il procedimento. La Figura **??** illustra i tempi di calcolo dell'euristica costruttiva base in funzione della dimensione n dell'insieme base. La prima osservazione è che i tempi di calcolo sono molto bassi (spesso "nulli" anche se misurati in microsecondi), il che non dà informazioni utili sulla loro dipendenza dalla dimensione. I pochi tempi non nulli crescono, ovviamente, con la dimensione stessa, ma con valori distribuiti su un ventaglio piuttosto ampio. Per valutare se questi risultati rivelano veramente una tendenza significativa,

- grafico tempi: non si vede praticamente nulla (tempi troppo bassi)
- grafico gap rispetto all'ottimo: si può fare il diagramma SQD, ma non ha molto senso, dato che le dimensioni sono eterogenee; d'altra parte ci sono solo 4 istanze per ogni dimensione

Istanza	f	f^*	δ
matrizn100m10	333	306	8.11%
matrizn100m20	1195	1137	4.85%
matrizn100m30	2457	2343	4.64%
matrizn100m40	4142	4055	2.10%
matrizn200m20	1247	1177	5.61%
matrizn200m40	4450	4333	2.63%
matrizn200m60	9437	9263	1.84%
matrizn200m80	16225	16046	1.10%
matrizn300m30	2694	2628	2.45%
matrizn300m60	9689	9449	2.48%
matrizn300m90	20743	20542	0.97%
matrizn300m120	35881	35592	0.81%
matrizn400m40	4658	4531	2.73%
matrizn400m80	16956	16653	1.79%
matrizn400m120	36317	35937	1.05%
matrizn400m160	62487	62091	0.63%
matrizn500m50	7141	6917	3.14%
matrizn500m100	26258	25605	2.49%
matrizn500m150	56572	55936	1.12%
matrizn500m200	97344	96543	0.82%

Nemmeno una delle istanze viene risolta all'ottimo. Un'analisi superficiale si fermerebbe qui.

Invece si può notare che tutte le soluzioni contengono il punto 1. Questo è strano, a prima vista. Ma non lo è se si osserva come funziona l'algoritmo costruttivo.

Nella prima iterazione del ciclo, quando si sceglie il primo punto della soluzione, tutti i punti `i` sono equivalenti, perché danno luogo ad una soluzione di valore nullo (quando c'è un punto solo, la somma delle distanze reciproche è nulla). Ovviamente, qualcosa non va nel criterio di scelta φ, per lo meno al primo passo.

Idee alternative:

- cambiare criterio di scelta in generale: ma quale usare?
- cambiare il criterio di scelta al primo passo:
  - scegliere il primo punto non in base a $f(x \cup \{i\})$, ma in base al fatto che sia un punto lontano dagli altri

$$\varphi(x,i) = \begin{cases} \sum_{j \in N} d_{ij} \text{ quando } x = \emptyset \\ f(x \cup \{i\}) \text{ quando } x \neq \emptyset \end{cases}$$

  - scegliere i primi due punti in modo che siano i più lontani fra loro; questo equivale (di solito, non rigorosamente sempre) a scegliere il primo punto come uno dei due più lontani fra loro e il secondo punto come il più lontano dal primo, cioè secondo la regola generale

$$\varphi(x,i) = \begin{cases} \max_{j \in N} d_{ij} \text{ quando } x = \emptyset \\ f(x \cup \{i\}) \text{ quando } x \neq \emptyset \end{cases}$$

- usare il primo punto come parametro dell'algoritmo e ripeterlo n volte, scegliendo ogni volta un punto diverso come primo punto della soluzione

Queste soluzioni costano tutte un po' di più di quella base in termini temporali. L'ultima, in particolare, costa n volte tanto, mentre le precedenti sommano solo un termine iniziale aggiuntivo, che non aumenta la complessità asintotica. Inoltre, è chiaro che l'ultimo algoritmo contiene in sé i precedenti, dato che esegue ogni possibile scelta del primo punto. Quindi li domina necessariamente.

- i diagrammi SQD rimangono poco significativi in sé, ma certamente significativi per il confronto fra gli algoritmi, dato che si riferiscono alla stessa popolazione di istanze
- i test di Wilcoxon fra le diverse varianti certamente confermano la dominanza dell'ultimo algoritmo, e probabilmente mostrano una prevalenza statistica delle inizializzazioni ragionevoli su quella base, ma cosa dicono fra le due varianti intermedie?
- i tempi di calcolo mostrano che l'ultima variante non è confrontabile davvero con le intermedie, ma cosa dicono delle intermedie rispetto a quella iniziale?
- e se si provassero, anziché n punti, solo i migliori rispetto a uno o all'altro criterio? se si procedesse nell'ordine, salvando tempi e risultati ottenuti via via, si potrebbe costruire un diagramma SQD temporizzato, che mostra il progresso dell'algoritmo nel tempo. Questo dovrebbe essere fatto a dimensione fissata, o meglio ancora per ciascuna istanza fissata

Con la versione che prova tutti i punti, si puo' fare lo studio dei tempi di calcolo, per valutare le costanti asintotiche. Occorre un diagramma semilogaritmico, dato che il tempo e' polinomiale. Il diagramma mostra una nuvola non del tutto lineare: il motivo è che n non è l'unico parametro in gioco, ma c'è anche k, e ogni istanza sta a sé. Servirebbero molte istanze per ogni coppia (n, k). Non le abbiamo, ma tracciamo lo stesso il diagramma e facciamo lo studio.

Ora bisognerebbe analizzare a fondo il trucco di salvare a parte in un vettore le distanze totali di ogni punto dalla soluzione corrente x. Questo sostituisce alla funzione che le calcola una funzione che le legge, ma comporta l'introduzione di un aggiornamento nella funzione `add_point_in`. Quindi si risparmia un tempo $O(|x|(n-|x|))$ per ogni nuova aggiunta, ma si perderebbe $O(n-|x|)$ in aggiornamento, oppure $O(n)$ se si aggiorna il campo anche nei punti interni alla soluzione. IL risparmio in tempo di calcolo è probabilmente invisibile nell'euristiche semplici, ma si dovrebbe vedere su quella che prova ogni punto di partenza. Oppure si possono usare le istanze da $n = 2\,000$ punti.

Altre euristiche costruttive? Siccome ogni punto alla fine si relazionera' con altri $k-1$, si potrebbe stimare il contributo che da all'obiettivo considerando non solo la distanza da x, ma anche la distanza dai $k-|x|-1$ altri punti ancora incogniti. Siccome tali punti saranno tendenzialmente lontani, si potrebbero prendere quelli con somma massima fra la distanza da x e la distanza dai $k-|x|-1$ punti più lontani. Comporterebbe di gestire ordinamenti di distanze, e aggiornarli ad ogni passo.

E le euristiche distruttive? Si tratta di partire con $x = N$ e poi togliere via via il punto con distanza minima dagli altri. Qui non ci sarebbe il problema dell'inizializzazione. Pero' i tempi di calcolo sono certamente maggiori e i risultati probabilmente peggiori, perche' il numero dei passi e' $n-|x|$ anziché $|x|$, dunque molto maggiore, e quindi maggiori sono le probabilita' di sbagliare. Nelle istanze con

$k < n/2$, questo dovrebbe portare a risultati migliori con le euristiche costruttive che con quelle distruttive. Ovviamente, l'ipotesi andrebbe testata sperimentalmente. Si potrebbero anche modificare le istanze date (banale: param m :=) e vedere che cosa succede.