01 - Meta Heuristics

Algoritmo Euristico. Si dice algoritmo euristico un algoritmo utilizzato generalmente per risolvere un problema NP-hard, che non fornisce una soluzione ottima al problema ma fornisce una soluzione sufficientemente buona in un lasso di tempo accettabile. Non esplora tutte le soluzioni possibili e viene guidato da scelte fatte sulla base della soluzione trovata finora. Tipicamente un algoritmo euristico è caratterizzato anche da una treshold, che può essere sia di tempo, di numero di iterazioni o di epsilon sulla soluzione trovata: questo aspetto permette di garantire che la soluzione trovata si trova ad una certa distanza massima rispetto alla soluzione effettiva (upper bound). Esistono vari tipi di algoritmi euristici:

- ***constructive***, ovvero gli algoritmi greedy, che prendono la scelta considerata migliore ad ogni iterazione, senza badare allo stato globale. Costruiscono un pezzo della soluzione ad ogni passo. Questo tipo di algoritmi è solitamente molto semplice e trova una soluzione feasible, anche se magari non è quella ottima
- ***local search***, si tratta di una classe di algoritmi che sono caratterizzati dall'esplorazione iterativa di un neighborhood di una soluzione. Si comincia a partire da una soluzione feasible e l'algoritmo continua finchè non si ottengono miglioramenti esplorando il neighbourhood: permette quindi di trovare soluzione ottime locali
- ***metaeuristiche***, è una classe di algoritmi che può essere considerata come l'evoluzione dei local search. Difatti questi algoritmi sono mirati ad una esplorazione più globale, inserendo i seguenti concetti:
    - *intensificazione*, ovvero il cercare una soluzione nel neighborhood della soluzione attuale
    - *diversificazione*, ovvero il cercare una soluzione lontana.

  Talvolta le metaeuristiche accettano anche un peggioramento della soluzione attuale, per cercare di scappare da ottimi locali e cercare un altro ottimo. Molto spesso si basano su meccanismi della natura.

Constructive heuristics. Alcuni esempi di euristiche costruttive possono essere date dai seguenti algoritmi:

- TSP con scelta del prossimo vertice da esplorare come il vertice più vicino al vertice attualmente considerato. Questo chiaramente potrebbe non portare alla soluzione ottima ma è un greedy molto semplice
- bin packing con first fit. In questo problema si ha a disposizione un certo numero di item e si deve calcolare il minimo numero di bin necessari per inserire tutti gli item. Ogni bin ha la stessa capacità. L'approccio first fit è di tipo greedy perchè quello che fa è scorrere la lista di item e inserire ogni item nel primo bin disponibile che abbia ancora abbastanza capacità per ospitare quell'item, senza badare al fatto che magari inserendo un item in un altro bin si sarebbero potuti inserirne meglio gli altri e minimizzare il numero di bin alla fine. Anche questo algoritmo quindi è molto semplice e non ha il concetto di esplorazione globale, bensì prende la decisione considerata migliore in quell'iterazione

Neighborhood. Dato uno spazio di soluzioni X e data una soluzione x appartenente a X. Si definisce neighborhood di una soluzione x, l'insieme delle soluzioni generate a partire da x tramite una singola move, tali che l'insieme delle soluzioni generate X' è un sottoinsieme di X (ovvero tutte le soluzioni generate sono valide e rientrano nell'insieme delle soluzioni possibili). Una move è definita come un'operazione atomica che assume un significato diverso in base al problema effettivo che si sta considerando.

Local search. Sono algoritmi caratterizzati dall'esplorazione di un neighborhood a partire da una soluzione iniziale. Di seguito, alcuni algoritmi di local search con il rispettivo significato di "move" in relazione a quel dato problema:

- ***TSP 2-opt.*** In questa versione del TSP si parte da una soluzione feasible, che in questo caso corrisponde ad un tour di città valido, e si considerano tutti gli archi (in una sola direzione, quindi ogni arco è descritto da una coppia [i,j] con i < j). Si considerano due archi e si verifica se la lunghezza totale del percorso varia in caso di swap del secondo nodo del primo arco e del primo del secondo arco. Si rimuovono quindi due archi e si ricollegano in modo diverso. In questo caso il costo per verificare le possibili combinazioni tra tutte le coppie di archi è di n(n-1) quindi $O(n^2)$ da sostenere ad ogni iterazione
    - *move*: rimozione di 2 archi e swap dell'arco "centrale"
- ***TSP 3-opt.*** In questa versione del TSP si fa quello che si fa col 2-opt, ma in questo caso si rimuovono 3 archi e quindi devono essere considerate tutte le possibili combinazioni tra 3 archi. In questo caso quindi il costo per verificare le combinazioni possibili per le triple di archi è di n(n-1)(n-2) quindi $O(n^3)$, da sostenere ad ogni iterazione.
    - *move*: rimozione di 3 archi e verifica del percorso minore ottenibile
- ***TSP or-opt.*** Questa versione del TSP si caratterizza per il fatto che si raggruppano da 1 a 3 città diverse e si prova a spostare l'intero gruppo (non modificando gli archi tra di loro) da un punto all'altro del tour. Questo processo viene ripetuto iterativamente e permette di trovare soluzioni migliori del 2-opt avendo comunque lo stesso costo computazionale $O(n^2)$.
    - *move*: spostamento di un gruppo di 1-3 nodi da un punto all'altro del tour
- ***insert move.*** Non rappresenta un algoritmo a sé, bensì un possibile modo generale per effettuare una move. Nello scheduling di task su diversi core ad esempio si può prendere un task e inserirlo su un altro core per verificare se il tempo totale di esecuzione diminuisce
    - *move*: spostamento di un item da un punto all'altro della soluzione
- ***swap move***. Non rappresenta un algoritmo a sé, bensì un possibile modo generale per effettuare una move. Nello scheduling di task su diversi core ad esempio si possono scambiare due task per verificare se il tempo totale di esecuzione diminuisce
    - *move*: scambio di due item della soluzione

In generale un algoritmo di local search si comporta nel seguente modo.

Local Search

1. Generate an initial solution $\mathbf{x} \in X$; $continue := true$
2. **while** $continue$ **do**
3. Find $\mathbf{x}' : f(\mathbf{x}') = \min\{f(\hat{\mathbf{x}}), \hat{\mathbf{x}} \in N(\mathbf{x})\}$
4. **if** $f(\mathbf{x}') < f(\mathbf{x})$ **then** set $\mathbf{x} := \mathbf{x}'$
5. **else** $continue := false$

In cui:

- 1: si genera una soluzione iniziale casuale all'interno dello spazio delle soluzioni
- 3: si trova x' tale che f(x') sia il minimo ottenibile a partire da N(x). Questo passaggio rappresenta la vera e propria esplorazione del neighborhood
- 5: l'algoritmo si interrompe non appena non viene trovata una soluzione migliorativa. Questo significa che si è arrivati in un ottimo locale

Un possibile miglioramento del local search è dato dal ***multistart*** che consiste nel lanciare diverse istanze diverse di questo algoritmo a partire però da soluzioni distanti e ben distribuite nello spazio delle soluzioni: in questo modo si trovano più ottimi locali aumentando la possibilità di trovare una soluzione migliore e comunque più vicina all'ottimo globale.

Metaeuristiche

Si esplorano di seguito le principali metaeuristiche esistenti. Come detto inizialmente, le metaeuristiche sono spesso basate su meccanismi della natura e hanno la caratteristica di esplorare più soluzioni invece che bloccarsi in un ottimo locale come avviene invece per la local search. Si sottolinea che le metaeuristiche non rappresentano algoritmi a sé stanti, bensì sono metodologie applicabili alla risoluzione di problemi noti come ad esempio il TSP. A differenza degli algoritmi di local search visti prima, che erano applicati ad un problema specifico (ad esempio il 2-opt TSP o il 3-opt TSP), le metaeuristiche forniscono un modello sulla base del quale costruire un algoritmo specifico per ogni diverso problema.
Le metaeuristiche, inoltre, si dividono in due diverse categorie, in base al modo di cercare la soluzione:

- ***population based.*** Sono ad esempio gli algoritmi genetici. Sono caratterizzati dal concetto di popolazione, che consiste in un gruppo esteso di diverse soluzioni. Seguono il meccanismo dell'evoluzione naturale e quindi creano nuove soluzioni a partire dall'attuale popolazione effettuando operazioni di mix (*crossover*) tra più soluzioni, modificando soluzioni generate tramite *mutazioni* o eventualmente modificando la popolazione inserendo altre soluzioni non generate a partire da quelle precedenti (*immigrazione*). Ogni soluzione in questo caso prende il nome di cromosoma. La popolazione viene scremata ad ogni iterazione in base alla *fitness*, che è un parametro che descrive quanto un cromosoma è "buono" nel senso che è adatto a sopravvivere fino alla prossima generazione.
    - vantaggio: molto abile ad esplorare un vasto spazio di soluzioni

    - svantaggio: non da molto peso alla densificazione dell'esplorazione intorno ad una singola soluzione, almeno che non siano presenti meccanismi quali l'hybridization che applica una local search su un cromosoma generato in modo da migliorarlo
- ***single solution/trajectory.*** Si contrappone agli algoritmi genetici. Questa classe di metaeuristiche parte da una soluzione ammissibile e ne cerca un'altra a partire da quella ed esplorando le soluzioni vicine e talvolta diversificando (come si fa nel simulated annealing quando si prende una soluzione peggiorativa).
    - vantaggio: algoritmi costruiti in questo modo arrivano prima a convergenza e si rivelano molto utili in caso di soluzioni ottime ravvicinate tra loro all'interno dello spazio delle soluzioni.
    - svantaggio: rischio di incappare in ottimi locali maggiore e si esplorano meno soluzioni

Population based

Si presenta di seguito un esempio di metaeuristica basata sul concetto di popolazione, ovvero un algoritmo genetico.

Un algoritmo genetico dotato di hybridization, esegue i seguenti passi, una volta stabilita una prima popolazione di partenza P e valutato la fitness con una specifica funzione che assegna ad ogni individuo in P un certo valore:

1. ***generazione di un insieme di genitori G*** sottoinsieme di P x P (non tutti gli individui della popolazione sono dei genitori)
2. ***crossover***, ovvero generazione di nuovi cromosomi effettuando operazioni di crossover tra coppie di genitori. Le soluzioni, rappresentate come vettori, possono essere crossate in più punti, si parla ad esempio di *crossover singolo* se il vettore viene diviso in un unico punto, di *crossover doppio* se viene diviso in due punti. Si genera una popolazione generata P_G
3. ***mutazione***, ovvero modifica casuale dei cromosomi generati P_G
4. ***hybridization***, ovvero applicare euristiche ad esempio local search per migliorare cromosomi generati P_G. Questo processo permette di velocizzare la convergenza
5. ***fitness evaluation*** di P_G
6. ***selection della P della prossima iterazione*** considerando la fitness della popolazione corrente formata sia da P che da P_G. Oltre agli individui strettamente migliori si potrebbero selezionare anche individui molto diversi dai migliori in modo tale da applicare diversificazione.

Si esegue questo loop fino alla condizione di terminazione.
Si sottolinea che al passo 2, è possibile che vengano generate soluzioni non feasible, per cui possono essere attuate due strategie:

- la più semplice, eliminarle direttamente da P
- eseguire semplici algoritmi per riparare le soluzioni non feasible. Ad esempio nel TSP se è stato generato un tour non ammissibile si applicano algoritmi semplici per la generazione di un tour ammissibile (seppur non ottimo ovviamente)

Single solution/trajectory

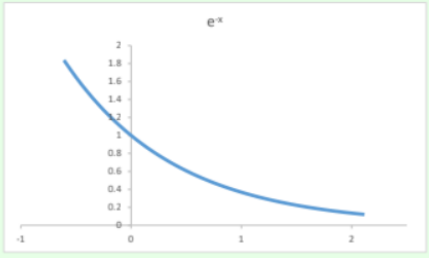
Si presentano in questa sezione varie metaeuristiche a single solution.

Simulated annealing. Si basa sul processo di annealing eseguito in metallurgia. In questo processo si riscalda un metallo fino alla temperatura di fusione, in cui l'energia tra gli atomi è altissima e infatti non sono stabili (in stato liquido). Successivamente si fa diminuire la temperatura lentamente in modo da dare tempo agli atomi di disporsi nel modo più ottimale possibile e creare alla fine un metallo la cui disposizione degli atomi allo stato solido è estremamente robusta. In caso di raffreddamento troppo repentino gli atomi arriverebbero comunque in uno stato solido ma avendo una struttura più disordinata e meno robusta.
Si può estendere questa meccanica anche per creare un metaeuristica, in particolare:

1. si sceglie una temperatura di partenza t_k (per simulare il processo di riscaldamento) e una soluzione feasible di partenza x
2. ad ogni iterazione si esplora un punto casuale contenuto in N(x),
    a. se il nuovo stato è migliore di quello precedente allora si accetta sicuramente
    b. se il nuovo stato è peggiore allora si accetta con una probabilità che dipende dalla distanza tra i due stati e dalla temperatura attuale. Il metodo per decidere se accettare o meno una nuova soluzione peggiorativa è chiamato *metodo monte carlo*
3. la temperatura viene decrementata. In questo modo (che simula la fase di raffreddamento) si diminuisce la probabilità alla prossima iterazione che venga accettata una soluzione peggiorativa

Una soluzione peggiorativa viene accettata con la seguente formula, dove la costante di Boltzmann viene spesso assorbita dalla temperatura e settata a 1. E_i e E_j rappresentando due stati successivi

$$e^{\left(\frac{E_i - E_j}{k_B T}\right)}$$



Il processo termina una volta raggiunta la condizione di terminazione, che può corrispondere ad un certo numero di iterazioni, ad una temperatura, al valore di una soluzione, ecc…
Dato che la decisione viene presa solo in base alla differenza tra due stati e alla temperatura, si può dimostrare secondo le proprietà delle catene di Markov che se la temperatura diminuisce abbastanza lentamente allora la soluzione ottima verrà trovata con probabilità 1. In altre parole

se la temperatura diminuisce in modo infinitamente lento allora la probabilità di trovare l'ottimo tende a 1. Questo, intuitivamente, è dovuto al fatto che questa metaeuristica esplora molto quando t è alta e ha un approccio più greedy quando invece è bassa. Esplorando molto all'inizio aumenta la probabilità di "incanalarsi" verso la soluzione ottima.

Lo pseudo algoritmo è il seguente:

1. find an initial solution $\mathbf{x}$;
2. $k := 0$; $t_0 :=$ initialValue; $\mathbf{x}^* = \emptyset$
3. **while**(continuation criterion) **do**
4. **for** (number of iterations with same t_k) **do**:
5. randomly select a solution $\mathbf{x}' \in N(\mathbf{x})$;
6. **if** $(f(\mathbf{x}') < f(\mathbf{x}))$ **then** $\mathbf{x} := \mathbf{x}'$;
7. **else** set $\mathbf{x} := \mathbf{x}'$ with probability $e^{(f(\mathbf{x})-f(\mathbf{x}'))/t_k}$;
8. **if** $f(\mathbf{x}) < f(\mathbf{x}^*)$ **then** $\mathbf{x}^* := \mathbf{x}$;
9. **end for**
10. $k := k + 1$; define new t_k with $t_k < t_{k-1}$;
11. **end while**
12. **return**$(\mathbf{x}^*)$

In cui:

- 4: il numero di iterazioni con lo stesso t_k è un parametro fisso e spesso relativo alla size di N
- 10: Nelle applicazioni pratiche del simulated annealing, decade il fatto che si trova la soluzione ottima con probabilità 1. Questo perché sarebbe impraticabile effettuare un tale numero di iterazioni, per questo motivo solitamente si gestisce t_k come una funzione lineare del tipo $t_{k+1} = at_k$ dove a è solitamente un fattore tra 0.8 e 0.99. Questo meccanismo non assicura di trovare una soluzione ottima ma è sufficiente per avvicinarsi abbastanza ad essa.

Tabu search. Si tratta di una metaeuristica molto simile alla local search ma a differenza di quest'ultima è dotata di una memoria, che può essere limitata o illimitata, chiamata *tabu list* e che permette a questo metodo di registrate le soluzioni già visitate in modo tale da non incappare in loop. In particolare la memoria si rivela molto utile dal momento che questo metodo permette di accettare soluzioni peggiorative che si trovano nel neighborhood di una soluzione precedente x. Senza memoria è ovvio che x si troverebbe nel neighborhood della soluzione peggiorativa scelta e questo porterebbe ad un loop, che è il motivo per cui il local search non può operare in questo modo.

L'algoritmo utilizzato per questa metaeuristica è il seguente:

1. Find a starting solution $\mathbf{x}$;
2. set $\mathbf{x}^* = \mathbf{x}$, $TL = \emptyset$;
3. **while** (not termination condition)
4. Find $\mathbf{x}' \in N(\mathbf{x})$: $f(\mathbf{x}') = \min\{f(\tilde{\mathbf{x}}), \tilde{\mathbf{x}} \in N(\mathbf{x}), \tilde{\mathbf{x}} \notin TL\}$;
5. $TL = TL \cup \{\mathbf{x}\}$; $\mathbf{x} = \mathbf{x}'$;
6. **if** $f(\mathbf{x}) < f(\mathbf{x}^*)$ **then** $\mathbf{x}^* = \mathbf{x}$
7. **end while**
8. **return** $\mathbf{x}^*$

In cui:

- 4: trova un x' appartenente a N(x) tale che f(x') sia il minimo ottenibile tra tutti i punti nell'intorno di x, a patto che non siano già stati inseriti nella tabu list. Il fatto che ci sia la tabu list previene, ad esempio, il ritorno immediato all'ottimo locale dopo averlo abbandonato: dato che l'ottimo viene messo in tabu allora non si può tornare indietro e l'algoritmo deve cercare un altro percorso per trovare soluzioni migliori

Dal punto di vista dell'utilizzo di memoria e del costo computazionale la tabu list può rappresentare un problema, questo perché ospitando tutte le soluzioni già visitate il costo per la visita e il mantenimento di tale tabella diventerebbe insostenibile, soprattutto perchè va fatto ad ogni iterazione. Per questo motivo vengono attuati alcuni accorgimenti:

- al posto che memorizzare le soluzioni visitate nella loro interezza si memorizzano solamente alcuni attributi che accomunano più soluzioni, ad esempio la move che porta da una soluzione all'altra. Memorizzando la move si evita che la stessa move venga fatta più volte, ad esempio se nel TSP 2-opt si scambiano due archi, si memorizzano gli archi swappati e si evita che la stessa move venga rifatta a breve. Chiaramente memorizzando attributi piuttosto che l'intera soluzione si precludono alcune soluzioni che potrebbero essere visitate in futuro, potenzialmente anche migliori di quelle già trovate
- memorizzare una lista infinita consente di evitare del tutto doppie visite ma allo stesso tempo rappresenta un costo ingestibile, per questo motivo si può valutare di adottare una lista limitata implementata come una coda FIFO. Con questa soluzione, per altro, è possibile visitare nuove soluzioni che prima magari sono state precluse a causa dell'inserimento in lista di un attributo. Ovviamente, però, con questa implementazione è possibile visitare più volte la stessa soluzione

Altre caratteristiche da tenere in considerazione della tabu search sono:

- ***tenure***, che rappresenta la lunghezza della tabu list stessa espressa in numero di iterazioni. La tenure esprime il numero di iterazioni che una soluzione/attributo rimane nella tabu list prima di essere rimossa. Una tenure alta incentiva la diversification dato che si dovranno sempre esplorare soluzioni nuove siccome quelle appena esplorate rimangono nella lista per molto. Una tenure bassa incentiva invece l'intensification, perché rimuove presto dalla lista soluzioni appena visitate per cercarne altre nelle vicinanze. Può comunque essere applicato un approccio ibrido applicando entrambe le

tecniche in momenti diversi, ad esempio in una zona ritenuta favorevole si può intensificare.

- ***aspiration criteria***, rappresentano eccezioni alle regole. Ad esempio una soluzione x per cui f(x) < f(x*) potrebbe essere adottata anche se presente nella tabu list
- ***diversification***, rappresentano tecniche per diversificare la ricerca ad esempio multi start o adottare penalità per scoraggiare l'utilizzo di mosse frequenti

Variable Neighborhood Search/Descent. Si tratta di due metaeuristiche molto simili che applicano il concetto di esplorazione di diversi neighborhood a partire da una singola soluzione. Questa operazione serve a scappare da minimi locali, ad esempio per schedulare processi su più core si potrebbe essere raggiunto un ottimo locale con mosse di insert ma se si applicassero mosse di swap invece si riuscirebbe a ridurre ulteriormente il tempo totale. VNS e VND cercano proprio di fare questo seguendo approcci leggermente diversi. Di seguito l'algoritmo:

```
1.   find an initial solution x; x* = x;
2.   repeat
3.       k := 1:
4.       while (k < k_max) do
5.           randomly select a solution x' ∈ N_k(x);(shake)
6.           x := Local_Search(x',k);
7.           if  (f(x) < f(x*)) then
8.               x* := x; k := 1;
9.           else
10.              k := k + 1;
11.          end if
12.      end while
13.  until (stopping condition)
14.  return(x*)
```

In cui:

- 4: k_{max} è il numero di neighborhood che devono essere esplorati in una stessa iterazione
- 5: l'operazione di shaking, definita come il selezionare randomicamente una soluzione all'interno del neighborhood scelto, è presente solo in VNS
- 6: applica una local search a partire da x' spostandosi nel neighborhood k. Il risultato sarà quindi un ottimo locale
- 8: ogni volta che viene trovata una soluzione migliorativa si riparte a scorrere i neighborhood

Entrambi gli algoritmi sono accomunati dal fatto che, su una singola soluzione, viene eseguito un ciclo tra tutti i neighborhood disponibili. Essi vanno intesi come tutti i possibili tipi di mosse che si possono applicare in quel dato problema che si può risolvere. In caso si voglia applicare questa metaeuristica al TSP, ad esempio, possibile N sono dati dal 2-opt, 3-opt, or-opt. K

rappresenta proprio l'indice del N che si sta considerando. La differenza sostanziale tra le due varianti è la seguente:

- VNS ha un'accentuazione sulla diversification, perchè prima di applicare la local search su un neighborhood scelto, seleziona casualmente una soluzione x' appartenente a quel N. Questa operazione serve a "scuotere" la soluzione attuale
- VND ha un'accentuazione sull'intensification, perchè non applica l'operazione di shaking ma inizia la local search direttamente sulla soluzione corrente

Iterated Local Search. Si basa sull'idea che si può sfuggire a un minimo locale perturbando la soluzione trovata e riapplicando una local search. La perturbazione della soluzione trovata è ovviamente l'elemento chiave in questa metaeuristica e corrisponde a modificare in qualche modo la soluzione. Ad esempio, nel TSP, si potrebbe pensare di cambiare a caso l'ordine di alcune città del tour. L'algoritmo è il seguente:

```
1.   find an initial solution x;
2.   x* :=Local_Search(x);
3.   repeat
4.       x := Perturbation(x*, history)
5.       x :=Local_Search(x);
7.       if  (f(x) < f(x*)) then
             x* := x;
8.   until (stopping condition)
9.   return(x*)
```

In cui:

- 4: il parametro history della perturbazione è opzionale e rappresenta un insieme di dati che potrebbero essere utili all'esecuzione dell'algoritmo. Si potrebbe tenere ad esempio traccia delle soluzioni già visitate, delle move già fatte o del numero di iterazioni senza miglioramento e così via.

Ruin and Recreate. Si tratta di una metaeuristica che si basa sull'idea di trovare una soluzione che corrisponde ad un ottimo locale, distruggere una sua parte e successivamente cercare di ricostruire una nuova soluzione a partire da quella distrutta. Questa tecnica può anche essere implementata nell'ILS come metodo di perturbazione della soluzione. La soluzione ottenuta dalla distruzione di quella ottima è non feasible, a partire da questa si possono applicare algoritmi greedy per costruire una soluzione feasible e verificare successivamente se è migliore di quella avuta precedentemente. L'algoritmo è il seguente:

```
1.  find an initial solution x;
2.  x* := Local_Search(x);
3.  repeat
4.      x̄ := Destroy part of solution x (x̄ is no longer feasible)
5.      Apply a greedy like algorithm to reconstruct a
        feasible solution x from x̄
6.      x := Local_Search(x);
7.      if (f(x) < f(x*)) then
            x* := x;
8.  until (stopping condition)
9.  return(x*)
```

Equicut

Si tratta di un problema NP-hard da risolvere quindi mediante metaeuristiche. Il problema è il seguente: dato un grafo G = (V,E) con |V| pari, pesato sugli archi (a,b), ovvero pesi w(a,b) > 0 per ogni nodo appartenente al grafo; si trovi la partizione (A,B) di vertici tale che:

- |A| = |B|
- la somma dei pesi degli archi che collegano un nodo in A con un nodo in B deve essere minima e cioè in simboli

$$\sum_{i \in A, j \in B} w(i,j) \text{ è minima}$$

Per studiare come risolvere questo problema si tengano a mente i seguenti concetti, a partire da un vertice x appartenente alla partizione X:

- si dice $\mathbf{E_x}$ la somma dei pesi degli archi che partono da x e arrivano a vertici appartenenti all'altra partizione. Si parla quindi di somma dei costi esterni
- si dice $\mathbf{I_x}$ la somma dei pesi degli archi che partono da x e arrivano a vertici appartenenti alla partizione X, ovvero la stessa di x. Si parla di somma di costi interni
- si calcola $\mathbf{D_x}$ come la differenza tra E_x e I_x e indica il "desiderio" che il vertice x ha di lasciare la propria partizione. Questo perchè se D è maggiore di zero questo indica che la somma dei costi esterni è maggiore di quella dei costi interni, o viceversa se D è minore di zero. Spostare un vertice con D > 0 nell'altra partizione conviene perché questa mossa abbassa il costo esterno totale della partizione di provenienza e abbassa quindi anche il costo del taglio.

Esistono vari metodi per risolvere questo problema

Kernighan e Lin. Questo algoritmo introduce il concetto di guadagno definito come segue. Il guadagno dato dallo scambio di una coppia di due vertici è calcolato come la somma del D del primo vertice con la l D del secondo vertice a cui viene sottratto il doppio del peso dell'arco che collega i due vertici. Più in dettaglio si ha la seguente formula:

$$D_{a(j)} + D_{b(j)} - 2w(a(j), b(j))$$

dove

- $D_{a(j)}$ e $D_{b(j)}$ indicano i “desideri” di entrambi i vertici presi in considerazione (al passo j)
- 2w(a(j), b(j)) ha l’obiettivo di evitare di sovrastimare il guadagno. Questo perché in entrambi i D è stato considerato il peso dell’arco (a,b) come costo esterno, e questo fa sì che il guadagno aumenti. In realtà però invertendo i nodi questo costo esterno non cambierebbe per nessuna delle due partizioni dato che l’arco rimarrebbe esterno, per questo motivo non viene considerato

L’algoritmo esegue i seguenti passi, ripetuti finchè non si ottiene più nessun miglioramento:

1. creazione di una copia G’ del grafo attuale
2. for j = 1 to n/2-1
    a. individuazione in G’ di una coppia di vertici a(j) appartenente ad A e b(j) appartenente a B tale che il guadagno g(j) ottenuto da un eventuale scambio sia massimo. In pratica qui si stanno selezionando i vertici il cui scambio darebbe il guadagno massimo al cut, ovvero ridurrebbe di più il costo del taglio
    b. scambio effettivo dei due vertici selezionati sopra e aggiornamento di E, I, D per tutti gli altri vertici, questo perchè ovviamente lo scambio di due vertici provoca cambiamenti a catena a tutti gli altri vertici
    c. i due vertici a(j) e b(j) appena scambiati vengono fissati in modo tale da non permettere ulteriori spostamenti
3. individuare l’indice k per cui il guadagno totale risulta massimo. In pratica, selezionando la coppia che fa guadagnare di più ad ogni iterazione c’è la possibilità che ad una certa j il guadagno inizi a diventare negativo, quindi non ha senso scambiare quei vertici. Non è comunque detto che ad ogni iterazione j cali il guadagno, magari può anche aumentare perché si cambiano i parametri relativi ad ogni vertice. Per questo motivo la funzione in j non è monotona decrescente e c’è da trovare questo parametro k per cui la funzione guadagno è massima.
4. se il guadagno totale trovato al punto 3 è maggiore di zero allora si applicano gli scambi anche al grafo originale G (si scambiano i vertici fino al passo k) e si torna al punto 1, altrimenti ci si ferma qui.

Di seguito lo pseudo algoritmo:

repeat
Crea una copia G' del grafo attuale
for $j := 1$ **to** $n/2 - 1$ **do**
Individua in G' i vertici $a(j) \in A$ e $b(j) \in B$ tali che $g(j) = D_{a(j)} + D_{b(j)} - 2w(a(j), b(j))$ sia massimo;
Scambia $a(j)$ e $b(j)$ e aggiorna $E_i, I_i, D_i, \forall i$;
Fissa $a(j)$ e $b(j)$ impedendo ulteriori spostamenti;
end-for
Individua il valore k che massimizza $\gamma = \sum_{i=1}^{k} g(i)$;
if $\gamma > 0$, **then**
scambia i vertici $a(1), \ldots, a(k)$ e $b(1), \ldots, b(k)$ nel grafo originale
endif
until $(\gamma \leq 0)$

Simulated Annealing. Per applicare il simulated annealing al problema dell'equicut si fanno le seguenti considerazioni:

- il neighborhood è formato da tutte le soluzioni che si possono ottenere a partire da una soluzione corrente con un'unica move. La move in questione consiste nello spostamento di un singolo vertice da una partizione all'altra. Si considerano quindi anche soluzioni non ammissibili dato che ci si troverà sicuramente ad avere due partizioni di cardinalità diversa
- la funzione obiettivo comunque tiene conto dell'inammissibilità come fattore di penalizzazione e in particolare tiene conto della differenza di cardinalità tra le due partizioni come segue

$$cost(s) = w(\delta(S)) + \beta(|S| - |V \setminus S|)^2, \ \beta \geq 0$$

L'algoritmo specifico per questo problema è il seguente:

Scegli casualmente una soluzione di partenza s;
$T :=$ TSTART;
while (numero soluzioni accettate $>$ MINNSOL) **do**
  for ($n \times$ SIZE) **do**:
    Scegli un vicino $s' \in N(s)$;
    if $(cost(s') < cost(s))$ **then** $s := s'$;
    else $s := s'$ con probabilità $e^{(cost(s)-cost(s'))/T}$;
  end-for
  $T := T \times$ TFACTOR;
end-while
Restituisci miglior s

In cui:

- Il fattore di penalizzazione serve a far aumentare il costo e quindi a far diminuire la probabilità che una soluzione non ammissibile venga accettata, soprattutto in iterazioni molto in avanti (dato che T sarà basso)

Genetic Algorithm. Utilizzati in due versioni, entrambe accomunate da crossover single e mutazione standard ma si diversificano per la fitness:

- la prima versione valuta la fitness con una penalità esattamente come fatto dal simulated annealing
- la seconda versione elimina la penalità nella fitness e introduce un filtro greedy per riparare soluzioni inammisibili

Tabu Search. Anche questa applicabile in due versioni:

- versione 1:
  - N formato dalla move di un solo vertice da un insieme all'altro come per simulated annealing. Può essere ristretto in modo tale da non considerare mosse che provochino lo sbilanciamento di un certo parametro MAX-UNBALANCE, aggiornato dinamicamente
  - tabu list contiene iterazione in cui ogni nodo è stato inserito in una delle due partizioni. Si vieta lo spostamento di un vertice nuovamento all'altra partizione se non sono passate almeno tabu-tenure iterazioni
- versione 2:
  - N formato dalla move di scambio tra due vertici come avviene per algoritmo KL
  - tabu list formata dall'iterazione in cui due vertici sono stati scambiati, e anche qua non posso essere scambiati nuovamente per tabu-tenure iterazioni
  - tabu tenure cresce nelle fasi di ricerca peggioranti (per favorire diversificazione) e diminuisce in quelle miglioranti (perchè magari ci si sta avvicinando ad un ottimo e si preferisce intensificare)

GRASP. Metodo che utilizza un algoritmo greedy randomizzato per definire una soluzione iniziale e poi migliorarla con un local search fatto swappando coppie di vertici in A e B molto similmente a quanto fatto in KL. L'intero processo si ripete fino ad un criterio di stop.

02 - Mixed Integer Linear Programming

MIP. Si parla di Mixed Integer Programming problem per indicare una categoria di problemi di ottimizzazione in cui, nella formulazione del problema, compare almeno una variabile intera. La sua sottocategoria, MILP, indica invece problemi di tipo MIP che sono caratterizzati dall'avere tutti i vincoli lineari e anche la funzione obiettivo lineare. Un problema di ottimizzazione viene formulato come segue:

$$
\begin{aligned}
\underset{x}{\text{minimize}} \quad & \sum_{j=1}^{n} c_j x_j \\
\text{subject to} \quad & \sum_{j=1}^{n} A_{ij} x_j = b_i, \quad i = 1, \ldots, m, \\
& \ell_j \leq x_j \leq u_j, \quad j = 1, \ldots, n, \\
& \text{some or all } x_j \text{ integer}
\end{aligned}
$$

In cui:

- la funzione da minimizzare o massimizzare viene chiamata funzione obiettivo e viene scritta come la moltiplicazione tra il vettore dei coefficienti (traslato) e quello delle n incognite
- ogni problema è caratterizzato da m vincoli, ogni vincolo viene espresso come moltiplicazione tra la matrice dei coefficienti dei vincoli A e il vettore delle incognite x
- inoltre sono presenti anche dei vincoli di dominio per le variabili, che indicano eventuali restrizioni al valore che può avere una variabile

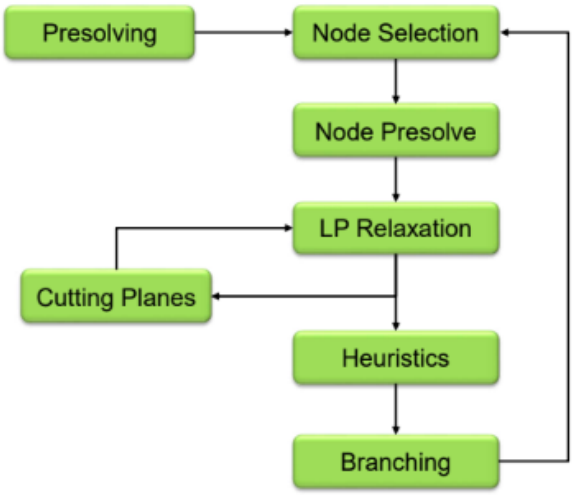
Fasi della risoluzione di un MIP. Per risolvere un problema di ottimizzazione di questo tipo, si alternano vari passaggi:

1. ***presolve***. In questa fase l'obiettivo è quello di eliminare eventuali ridondanze nella formulazione del problema. Questo si traduce nella riformulazione di eventuali vincoli ripetuti o nell'eliminazione di una o più incognite. Lo scopo di questa fase è quello di creare un problema equivalente che abbia meno vincoli e/o variabili in modo tale da velocizzare il processo di risoluzione vero e proprio. Oltre alla fase di identificazione di eventuali vincoli ridondanti e implicati a vicenda, si effettuano anche delle operazioni sul singolo vincolo indipendentemente dagli altri, detto single row reductions, che si suddividono in:
    a. *clean up rows*, che consistono nella rimozione di vincoli non necessari (ad es con coefficiente zero)

    b. *bound strengthening*, che consistono nel rafforzare i limiti imposti da un vincolo di dominio di una variabile sulla base dei vincoli del problema
    c. *coefficient strengthening,* che consiste nel rafforzare i limiti imposti dai vincoli per restringere il campo delle soluzioni ammissibili e renderlo più coincidente possibile con il campo delle soluzioni intere. In pratica si "tagliano" via porzioni di area ammissibile che però non sarebbe stata intera
2. ***risoluzione del rilassamento continuo***. In questa fase si trasforma il problema intero in un problema di linear programming rilassando i vincoli di integrità sulle variabili. La risoluzione di un problema di programmazione lineare avviene sempre in tempo polinomiale se ha soluzioni ammissibili (ad esempio tramite algoritmo del simplesso). Una volta trovata la soluzione ottima del rilassamento continuo si ottiene un bound sulla soluzione intera. In particolare in caso di problema di minimizzazione si ottiene un lower bound, si ottiene un upper bound altrimenti. Questo perché sicuramente la soluzione intera non sarà migliore di quella lineare, siccome quella lineare è proprio quella ottima. Se in un problema di massimo la soluzione ottima è frazionaria è intuitivo pensare che la soluzione ottima intera sarà al più grande quanto l'approssimazione per difetto della soluzione ottima continua.
3. ***piani di taglio***. In questa fase vengono applicati nuovi vincoli per tagliare fuori dalla regione ammissibile la soluzione lineare trovata al punto 2, senza però tagliare fuori eventuali soluzioni intere. Inserendo nuovi vincoli si crea un LP diverso che deve essere risolto nuovamente e quindi si torna al punto 2, per poi effettuare un nuovo taglio. Questa fase viene iterata finché non sono più disponibili tagli validi (ovvero non è più possibile escludere soluzioni frazionarie senza escluderne anche altre intere). L'obiettivo dei cutting planes è quello di restringere la regione ammissibile e diminuire i passaggi necessari al branch and bound eseguito al punto 4. In questa fase il bound trovato migliora ad ogni iterazione, quando non migliora più si passa alla fase successiva.
4. ***applicazione di euristiche***. In questa fase si cerca una soluzione ottima intera a partire dalla soluzione continua trovata al punto precedente.
5. ***branching variable selection***. In questa fase viene effettuato l'algoritmo del branch and bound. Quello che si fa è inserire nuovi vincoli che escludono il valore frazionario ottimo di una certa incognita. Se ad esempio è stata trovata x=3.5, il branch and bound viene eseguito risolvendo il rilassamento continuo dei problemi LP inserendo i vincoli x<=3 e x>=4. Una volta risolti entrambi i problemi si continua eventualmente l'esplorazione del ramo più "promettente" ovvero quello che non ha presentato soluzioni inammissibili o peggioranti rispetto al bound trovato. Un ramo la cui soluzione è peggiorante è un ramo in cui:
    a. *problemi di massimizzazione*: la soluzione del nodo corrente fornisce un UB minore del LB intero già trovato oppure fornisce una soluzione non feasible. In questo tipo di problemi ogni soluzione feasible intera trovata fornisce un LB valido
    b. *problemi di minimizzazione*: la soluzione del nodo corrente fornisce un LB maggiore del UB intero già trovato oppure fornisce una soluzione non feasible. In questo tipo di problemi ogni soluzione feasible intera trovata fornisce un UB valido

Quindi l'upper bound diminuisce a gradini e il lower bound aumenta in maniera continua in problemi di minimo. Discorso invertito in caso di problema di massimizzazione. Il gap tra lower bound e upper bound pari a zero indica che è stata trovata la soluzione ottima ma nei casi reali questo gap non è mai zero ma è abbastanza preciso da fornire una soluzione ottima adeguata.

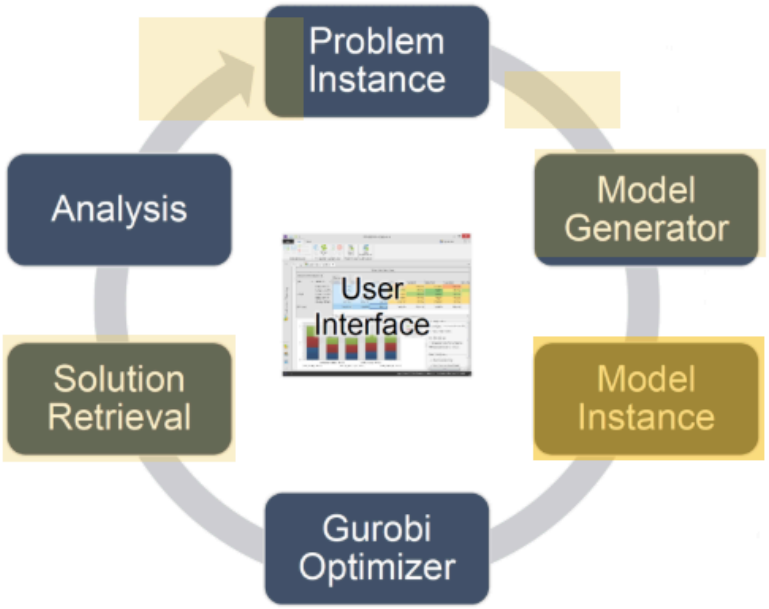
Il diagramma di sequenza per la risoluzione di un MIP è quindi il seguente:



Successivamente ad una fase di presolving iniziale si iterano le seguenti fasi:

1. presolve del problema (all'inizio è presente un singolo nodo, dato dal problema principale)
2. rilassamento continuo
3. cutting planes, che finché aggiunge tagli richiede la risoluzione del LP generato, finché non ci sono più tagli disponibili. Questo migliora il bound gradualmente (continuo)
4. heuristics, che generano una soluzione intera ammissibile. Questo fornisce un upper bound (intero)
5. branching. In questa fase si cerca di migliorare il bound intero a gradini. Il branching produce ogni volta 2 diversi problemi LP e quindi il ciclo si ripete dal punto 1.

Gurobi. Si tratta di un software risolutore di problemi di ottimizzazione. Per utilizzarlo, è necessario definire il problema da risolvere in maniera formale e comprensibile dal software, e successivamente, una volta ottenuta la soluzione, reiterare il ciclo fino a quando non si è ottenuta una soluzione sufficientemente accettabile. Il ciclo di modellazione di un problema in Gurobi è descritto dalla seguente immagine:



Le fasi descritte sono:

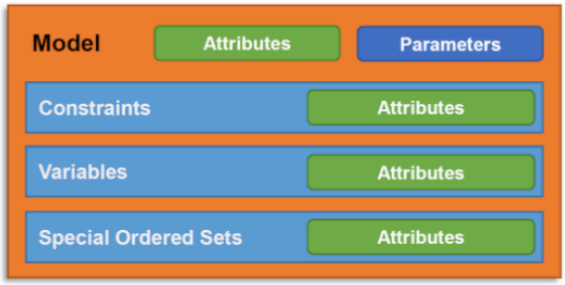
1. *problem instance*, formulazione del problema da risolvere, si fa su carta o comunque non si da ancora “in pasto” al software
2. *model generator*, traduzione del problema da risolvere in un linguaggio comprensibile dal software. In questa fase viene creato un modello, questo significa che il problema viene descritto tramite vincoli/funzione obiettivo parametrizzate.
3. *model instance*, in questa fase si settano effettivamente i parametri inseriti al punto precedente. Si può considerare la relazione tra fase 2 e fase 3 come la relazione tra definizione di una classe e instanziamento di un oggetto relativo a quella classe. Possono esserci quindi anche più istanze
4. *gurobi optimizer*, fase di vera e propria esecuzione dell’ottimizzazione
5. *solution retrieval*, si tratta di stabilire un metodo per ottenere la vera e propria soluzione a partire dai dati prodotti dal software, che talvolta possono essere di grandi dimensione
6. *analysis*, in cui si analizza la soluzione appena trovata e si riparte eventualmente dal punto 1

Si sottolinea che le fasi di cui sopra appartengono ad un ciclo siccome, soprattutto nel caso di un problema complesso, è improbabile trovare la soluzione che si desidera al primo tentativo: è comune quindi rimodellare il problema con i suoi vincoli e variabili più volte per ottenere una soluzione accettabile.

Dal punto di vista puramente tecnico, Gurobi può essere utilizzato sia da shell sia da API python sia da altri linguaggi. Tutti si interfacciano comunque con l’API interna scritta in C che è il vero e proprio core del software, che si occupa di interfacciarsi con i modelli e gestire l’esecuzione della risoluzione.
Un modello è caratterizzato da:

- *parametri*, che sono valori da settare utili puramente al software per regolare l'esecuzione della risoluzione. Ad esempio un parametro potrebbe essere il time limit massimo che ha per risolvere un'istanza
- *attributi*, che sono i valori effettivi dei parametri che si definiscono in fase di model generator. Sono quindi i rhs dei vincoli, lower bound delle variabili, ecc…

Inoltre, ogni modello eredita anche i parametri di ambiente. Un ambiente è un insieme di modelli che sono accomunati dagli stessi parametri globali.



03 - Exponential Constraints

TSP

Travelling Salesman Problem. Questo famoso problema di ottimizzazione consiste nel trovare il percorso più breve tra varie città. Formalmente, il problema del TSP consiste nel trovare il tour hamiltoniano più breve tra i nodi di un grafo.
Viene definito **tour hamiltoniano** la sequenza di archi che collega tutti i nodi del grafo, la sequenza inizia e termina nello stesso nodo e ogni nodo viene visitato una singola volta. Il TSP esiste in due varianti, entrambe NP-hard, il ATSP, ovvero TSP asimetrico, il cui grafo considerato è un grafo orientato; oppure STSP, con cui si riferisce al problema del TSP applicato ad un grafo non orientato.
Nonostante sia un problema formulato molto tempo fa esistono ancora oggi proposte di soluzioni nuove, ognuna con diverse garanzie. Ad esempio è stato fatto un algoritmo per TSP che garantisce le performance, ovvero un discostamento massimo garantito dalla soluzione ottima (ad es. una performance garantita 2 significa che la soluzione può essere al massimo il doppio di quella ottima)

Vincoli del ATSP. Un problema TSP su grafo orientato è caratterizzato dai seguenti vincoli che devono essere presi in considerazione in fase di modellazione del problema:

1. ci sono n archi
2. ogni vertice i ha esattamente un arco uscente
3. ogni vertice j ha esattamente un arco entrante
4. uno dei seguenti vincoli deve essere inoltre valido, per assicurare la presenza del tour hamiltoniano (vincolo di connettività):
    a. non esiste nessun circuito con meno di n archi (in questo modo si garantisce che ci sia un percorso chiuso tra tutti i vertici, rispettando ovviamente i vincoli sopra)
    b. per ogni partizione di vertici esiste almeno un arco che va da una partizione all'altra (se non esistesse, significherebbe che sono presenti almeno 2 circuiti distinti)

ASTP model. Il modello viene rappresentato come segue:

$$x_{ij} = \begin{cases} 1 & \text{if arc } (i,j) \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

$$u_i = \text{order of visit of node } i$$

$$\min z = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad i \in V$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad j \in V$$

$$x_{ij} \in \{0,1\} \quad (i,j) \in A$$

$$u_i \geq 0 \quad i \in V$$

- funzione obiettivo: minimizzare il costo del tour formato dagli archi selezionati (che vengono settati a 1 e fanno valere il rispettivo costo)
- vincoli:
    a. solo un arco che ha come vertice di partenza i deve essere settato a 1. Questo serve a soddisfare il vincolo 2, per cui ogni vertice deve avere solo un arco uscente
    b. solo un arco che ha come vertice di destinazione j deve essere settato a 1. Questo serve a soddisfare il vincolo 3, per cui ogni vertice deve avere solo un arco entrante

In base al fatto che venga considerato il vincolo 4a o 4b il problema può poi essere modellato secondo un altro vincolo, diverso per ogni modello:

$$u_1 = 0$$

$$u_j - u_i \geq 1 - M(1 - x_{ij}) \quad (i,j) \in A, j \neq 1$$

Questa disequazione soddisfa il vincolo 4a. In particolare questo vincolo afferma che:

- se esiste l'arco che collega il vertice i al vertice j allora bigM si annulla e il vincolo viene effettivamente attivato. Con questo vincolo si costringe il vertice j ad avere un valore di ordinamento pari a quello del vertice i + 1 (cioè i due vertici sono successivi nell'ordinamento)
- se l'arco non viene scelto, invece, bigM ha valore e rende il vincolo sempre vero (-infinito). In pratica il vincolo viene quindi disattivato se l'arco tra i due vertici non viene scelto

BigM ha proprio lo scopo di attuare il meccanismo di "switch" del vincolo, e corrisponde ad un valore molto grande.
Questa formulazione non è comunque applicabile nella pratica perché effettuando il rilassamento continuo delle variabili non si riuscirebbe ad applicare bene il meccanismo di switch, dato che bigM non verrebbe mai completamente annullata a causa del fatto che xij avrebbe un valore continuo da 0 a 1.

$$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 1 \quad S \subset V$$

Questa disequazione è la trascrizione diretta da quanto espresso dal vincolo 4b. In particolare questo vincolo afferma che la sommatoria degli archi scelti con vertice di partenza appartenente alla prima partizione S e vertice di destinazione appartenente ad una diversa partizione V\S, sia maggiore o uguale a 1. Ovvero deve esistere almeno un arco che va da una partizione all'altra.
Questa formulazione è difficilmente applicabile siccome ha complessità 2^n non so perchè.

$$\sum_{(i,j) \in A(S)} x_{ij} \leq |S| - 1 \quad S \subset V$$

$$A(S) = \{(i,j) \in A : i,j \in S\}$$

Questo è il vincolo di eliminazione dei subtour e soddisfa il vincolo 4b. Questa disequazione afferma che per ogni possibile partizione possibile creata a partire da A deve essere presente un numero di archi almeno inferiore di un'unità rispetto al numero di vertici in quella partizione. Questo implica, per via dei vincoli 2 e 3, che deve essere presente almeno un arco entrante e uno uscente dalla partizione. Se il numero di archi fosse uguale a quello dei vertici allora sarebbe sicuramente presente un circuito chiuso.
Anche questa formulazione è difficilmente applicabile a causa del numero di vincoli necessario che è esponenziale, per la precisione 2^n siccome deve esserci un vincolo per ogni possibile sottoinsieme di vertici (cioè per ogni partizione possibile).

STSP model. La formulazione del modello simmetrico per il TSP può essere condotta a partire dal modello asimmetrico, con le dovute accortezze:

- i costi $c_{i,j}$ e $c_{j,i}$ sono uguali siccome il grafo non è orientato e non cambia il costo di andata e di ritorno da un nodo

- per il motivo sopra, in caso si dovessero rilassare i vincoli sulla connettività allora verrebbero generati molti subtour di 2 vertici ciascuno siccome nella formulazione STSP è considerato un ATSP ma con una coppia di archi tra di loro, entrambi in direzioni opposte e con lo stesso costo.

In questo caso i vincoli sono leggermente diversi, in particolare:
1. sono presenti n archi
2. sono presenti 2 archi incidenti per ogni vertice (questo vincolo racchiude sia il 2 che il 3 del ATSP)
3. non può essere presente nessun circuito con meno di n archi (pari al vincolo di connettività del ATSP)

Il modello è il seguente:

$$x_e = \begin{cases} 1 & \text{if edge } e = \{i,j\} \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

$$\min z = \sum_{e \in E}^{n} c_e x_e$$

$$\sum_{e \in \delta(i)} x_e = 2 \quad i \in V$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \quad S \subset V$$

$$x_e \in \{0,1\}$$

$$\delta(S) = \{e = \{i,j\} : i \in S, j \in V \setminus S\}$$
$$E(S) = \{e = \{i,j\} \in E : i,j \in S\}$$

Rilassamento vincoli esponenziali. Alcuni problemi di ottimizzazione sono caratterizzati da un altissimo numero di vincoli, precisamente si tratta di un numero esponenziale come visto nei vincoli di connettività del problema del TSP. Difatti per verificare che non siano presenti sottotour è necessario verificare il vincolo per ogni possibile sottotour, che è quindi un numero esponenziale. Per evitare di fare questo può essere adottata la strategia seguente:
- si modella il problema eliminando i vincoli esponenziali
- si risolve il rilassamento continuo e si trova la soluzione ottima in seguito al processo di branch and bound/cut
- una volta trovata la soluzione ottima si verifica che rispetti anche i vincoli esponenziali tramite una certa funzione di separazione:
    - se la funzione di separazione da esito positivo, ovvero che i vincoli sono rispettati allora la soluzione intera trovata è quella ottima
    - altrimenti si ripete da capo l'intero ciclo aggiungendo però il vincolo che è stato infranto

La funzione di separazione descritta sopra deve avere il compito di riuscire a trovare almeno un vincolo che è stato infranto oppure deve poter dimostrare che nessun vincolo è stato infranto.

Questa funzione dovrebbe essere il più semplice possibile e non è scontato realizzarla. Nel problema del ATSP, nello specifico, la funzione di separazione è realizzata come segue:

Procedure find-subtour $G' = (V, A')$

comment: A' contains only subtours or a single tour
for each $i \in v$ **do** $visited(i) = false$
$smallest = n + 1$
for each $(i, j) \in A' : visited(i) = false$ **do**
  $isave = i,\ narcs = 1,\ visited(i) = true,\ Tour = \{i\}$
  repeat
    find $k : (j, k) \in A'$, set $visited(j) = true,\ Tour = Tour \cup \{j\}$
    $i = j,\ j = k,\ narcs = narcs + 1$
  until $isave = j$
  if $narcs < smallest$ **do**
    $smallestTour = Tour,\ smallest = narcs$
return $smallestTour$

In ordine, la procedura sopra esegue questi passi:

1. set del flag visited = false per tutti i vertici del grafo e set dello smallest tour a n+1
2. per ogni arco appartenente ad A' (quindi per ogni arco scelto e appartenente al tour), se non è ancora stato visitato:
    a. set delle variabili isave per ricordare il primo vertice, narcs per inizializzare il numero di archi appartenenti a quel tour, si imposta il vertice i come visitato e si mette nel tour che si sta costruendo
    b. poi, per ogni arco a partire da j (vertice di destinazione dell'arco a partire da i):
        i. seleziona l'arco scelto e imposta j come visitato e lo aggiunge al tour
        ii. aggiorna tutte le variabili necessarie per progredire la ricerca a partire da j
    c. il ciclo si ferma quando viene rilevato che il nodo di destinazione dell'arco che si sta considerando è uguale al vertice di partenza isave = i
    d. si registra il numero di archi del tour appena trovato

Lo scopo dell'algoritmo sopra è quello di restituire il tour con il minor numero di archi, questo ha il seguente significato:

- se il tour restituto ha n archi allora la soluzione è una soluzione valida
- altrimenti si introduce il vincolo che elimina il sottotour più breve. Viene eliminato solo quello più breve perché è una buona pratica e comunque eliminando quello la prossima soluzione sarà sicuramente diversa e magari comparirebbero altri subtour. Eliminando tutti i subtour trovati si rischierebbe di introdurre troppo vincoli che comunque hanno un costo per essere verificati e riducono l'efficienza

Lazy constraints. Il check effettuato dalla funzione sopra può essere applicato alla fine dell'esecuzione del metodo iterativo. Questo significa che si trova la soluzione ottima intera del problema generato a partire dal modello che si sta considerando attualmente; e solo successivamente si verifica che la soluzione sia feasible; se così non dovesse essere si aggiunge il vincolo al modello e si genera un altro problema che deve essere risolto da capo.

Applicando la tecnica delle lazy constraints, invece, la funzione sopra diventa un vero e proprio callback che il solver chiama non alla fine, bensì ogni volta che viene trovata una soluzione intera, anche non ottima. Così facendo il solver non deve essere riavviato più volte ma chiama la funzione ogni volta che trova una soluzione intera nel processo di branch and cut. Se la soluzione intera trovata non dovesse essere feasible si aggiunge praticamente un nuovo vincolo che stringe la regione ammissibile del problema (esattamente come se fosse un taglio), ma il sottoalbero viene comunque esplorato (dopo aver risolto nuovamente il nodo corrente con il vincolo aggiornato).

04 - Machine Learning and Mathematical Optimization

Negli ultimi anni si sta cercando di utilizzare simultaneamente sia il machine learning che l'ottimizzazione matematica.
L'ottimizzazione matematica si caratterizza per essere più rigorosa e precisa, nel senso che viene eseguita sulla base di una struttura precisa con i relativi vincoli.

- vantaggi: produce o almeno cerca di produrre una soluzione ottima
- svantaggi: ha tempi di computazione più alti e necessità di conoscere a fondo il problema da studiare.

Il machine learning, d'altro canto, cerca di risolvere problemi di ottimizzazione sulla base di ciò che ha imparato dai dati di altri problemi di ottimizzazione.

- vantaggi: non è necessario conoscere a fondo il problema siccome non è necessario definire una struttura specifica; presenta il vantaggio di essere più rapido nell'esecuzione una volta superata la fase di training
- svantaggi: soluzioni meno precise o comunque corrette con una certa probabilità; fatica inoltre a trovare soluzioni per nuovi problemi non ancora noti.

Il machine learning è un insieme di tecniche mirate all'imparare una serie di pattern e strutture utili per classificare, predire o ultimamente anche generare, dati. Il machine learning impara dai dati e ad ogni dato viene abbinato un insieme di caratteristiche dette features, utili per memorizzare il pattern/struttura di quel dato.
Un algoritmo di machine learning è caratterizzato da un prima fase di learning a partire dai dati forniti in input in modo da creare una funzione non lineare utile a classificare i dati che poi verranno sottoposti in input nella fase posteriore al learning. Il learning può essere di due tipi:

- ***supervised/imitation.*** In questo tipo di learning l'algoritmo riceve in input dati già etichettati da un esperto e l'obiettivo di questa fase è che l'algoritmo riesca, una volta allenato, ad emulare il comportamento dell'esperto per quei dati. Lo svantaggio qui è che c'è rischio di **overfitting**, ovvero un algoritmo impara molto bene a lavorare con un insieme specifico di dati, rimanendo poco flessibile in caso venga sottoposto ad un nuovo dataset, producendo spesso risultati sbagliati. Inoltre tramite imitation il modello prodotto non può superare la conoscenza dell'esperto, proprio perché non saprebbe classificare nuovi dati in maniera giusta.
- ***reinforcement***. In questo tipo di learning l'algoritmo viene trattato come agente in un ambiente. L'agente può effettuare una determinata azione in base alla propria policy (strategia per massimizzare le ricompense) e in base allo stato dell'ambiente. L'ambiente

registra l'azione dell'agente e cambia il suo stato di conseguenza. A questo punto l'ambiente comunica la transizione di stato all'agente, oltre ad una ricompensa che serve a comunicare all'agente quanto la sua azione è stata giusta. Con questo metodo si favorisce la **generalizzazione**, per cui il modello riesce a classificare dati su cui non si è mai allenato.

Nella fase successiva al learning, chiamata validation, si valuta l'effettiva qualità della funzione imparata rispetto ai dati su cui il modello è stato allenato.

Machine learning e ottimizzazione matematica possono essere combinati in tre diversi modi:

- ***end-to-end***. In questo metodo il machine learning sostituisce completamente l'ottimizzazione matematica. Il modello di machine learning impara da problemi con le rispettive soluzioni e cerca di trovare una soluzione sulla base del problema fornito in input.
    - vantaggio: veloce a trovare la soluzione a tempo di esecuzione
    - svantaggi: elevato rischio di overfitting, difficoltà a trovare soluzioni a problemi molto diversi da quelli visti in fase di training
- ***learning properties***. Questo metodo prevede l'effettiva combinazione di machine learning e ottimizzazione matematica. Questo perché il modello di machine learning cerca di trovare una soluzione di buona qualità da dare poi in pasto all'ottimizzazione (ad esempio un buon upper bound per iniziare poi un branch and bound).
    - vantaggio: migliora efficienza senza sacrificare qualità dato che può essere integrato in metodi euristici già esistenti
- ***repeated decisions***. Questo metodo chiama in causa il machine learning più volte durante l'ottimizzazione ed è un metodo più complesso degli altri

Clustering

Si definisce clustering l'operazione mediante la quale si cerca di dividere un dataset in più classi sulla base della similarità tra i punti-dato contenuti nel dataset. La clusterizzazione è un processo che porta ad avere un dataset partizionato in più classi (e talvolta permette anche di calcolare il numero di classi totali a cui un dato può appartenere ma k-means invece ha bisogno di questo dato in input). I punti presenti in ogni cluster sono molto simili tra loro, i punti di cluster diversi invece sono poco simili tra loro. L'operazione di clusterizzazione avviene senza l'impiego di alcuna label.

Il concetto di similarità è definito come l'inverso della distanza. Si dice che due punti hanno distanza zero se coincidono e se quindi hanno le stesse caratteristiche.

Il clustering si divide in due diversi tipi:

- ***partitioning*****,** se l'obiettivo è quello di dividere il dataset in più partizioni completamente distinte
- ***hierarchical***, se l'obiettivo è quello di dividere il dataset in partizione che formano una gerarchia. Con questo si intende che si forma un albero di partizioni in cui le due partizioni figlie formano una partizione padre, e così via fino al nodo radice dell'albero.

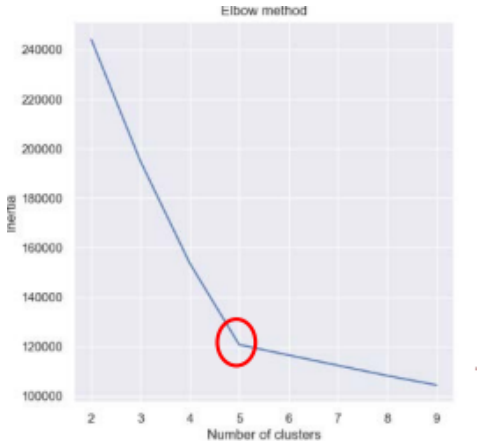
K-means. Si tratta di un algoritmo utilizzato per dividere i dati in cluster. Lo scopo di questo algoritmo è quello di dividere il dataset in k clusters, dove k deve essere il numero di classi che riduce il più possibile l'inerzia del dataset.
L'inerzia viene definita come la somma della radice quadrata delle distanza di ogni punto del dataset dal centroide del cluster di appartenenza. Quindi è chiaro che l'inerzia viene minimizzata da k uguale al numero di punti del dataset, ma questo chiaramente non è una buona soluzione non ci sarebbe alcun margine di errore nei cluster e sarebbe una classificazione per niente flessibile. L'inerzia viene calcolata come segue, dove x è il punto i e u è il centroide del cluster.:

$$inertia = \sum_{i=1}^{n} \min_{j \in \{0, 1, .., k-1\}} (||x_i - \mu_j||^2)$$

L'algoritmo K-means esegue i seguenti passaggi:

1. inizializzare k cluster (k è in input) scegliendo k punti randomicamente
2. ripetere iterativamente:
    a. dividere il dataset in modo tale da mettere ogni punto nel cluster la cui distanza dal centroide è minore
    b. ricalcolare il centroide di ogni cluster in modo da fare una media delle distanza dei punti contenuti al suo interno
3. ripetere finché nessun punto cambia cluster

Si noti che in questo algoritmo è necessario impostare un numero k di cluster, per farlo si segue l'elbow method che consiste nell'eseguire l'algoritmo più volte con un numero k di cluster sempre crescente e registrare i valori di inerzia. Ad una prima fase in cui aumentando k l'inerzia scende velocemente ne segue una in cui anche aumentando k l'inerzia rallenta la sua decrescita. Di solito si sceglie il punto in cui l'inerzia inizia a rallentare la decrescita come buon punto di trade off.



Hierarchical clustering. Questo tipo di clustering può essere di due diversi tipi:

- ***agglomerative***, che è un approccio bottom up e che unisce tra di loro coppie di cluster fino ad arrivare al cluster "radice". Per eseguire questo tipo di clustering si piazza quindi ogni oggetto in un cluster e poi si valutano i cluster da unire in base alla distanza tra un cluster e l'altro. Il concetto di distanza punto-cluster e cluster-cluster può essere definito in più metodi (si può considerare la distanza minima tra due dati in due cluster diversi, o

quella massima; si può considerare la distanza media tra tutte le coppie di punti contenute nei due cluster…)

- ***divisive***, che è un metodo top-down che invece parte da un unico cluster e divide ricorsivamente i cluster