Università di Torino
Dipartimento di Informatica
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Corso
"*Algoritmi e Complessità*"

Programma didattico

Luca Roversi

26 maggio 2022 (A.A.21/22)

Indice

Introduzione e Problemi motivazionali 1

Informazioni introduttive

"Meta"-introduzione.mp4 al corso con considerazioni di carattere generale.

Introduzione.mp4 è più centrato su dettagli organizzativi, in accordo col documento Introduzione.pdf.

1.1 Problemi (essenzialmente) reali

Poniamoci l'obiettivo di automatizzare la soluzione a tre problemi Valutazioni, Bando e Rischio, che individuiamo sin d'ora come "motivazionali" per questo corso.

- Valutazioni.pdf richiede di estrarre la migliore valutazione possibile da test di una certa complessità che possono distribuire un punteggio totale maggiore di quello ammesso come votazione finale.

  Valutazioni.mp4 illustra i dettagli con un esempio.

- Bando.pdf richiede di scegliere quali proposte di progetto finanziare interamente, cioè esattamente secondo l'ammontare richiesto, tra quelle pervenute in relazione a tre aree di intervento, tenendo conto di limiti totali di finanziamento e di valutazioni sull'utilità di ogni proposta progettuale.

  Bando.mp4 illustra i dettagli con un esempio.

- Rischio.pdf richiede di stabilire quali richieste di credito accordare interamente, cioè senza riduzioni di quanto richiesto, entro un limite totale prefissato di disponibilità, avendo due obiettivi:

  – minimizzare un parametro di rischio che dipende sia dall'affidabilità stimata del richiedente credito, sia dall'ammontare della richiesta di finanziamento;

  – massimizzare il profitto derivante dal totale dei prestiti.

Rischio.mp4 illustra i dettagli con un esempio.

1.1.1 Osservazioni

Per chi non sia alle prime armi nel dover risolvere problemi computazionali, posto di fronte ad un problema come Valutazioni, Bando o Rischio, è naturale porre l'attenzione su alcuni aspetti che qui illustriamo.

Per esempio, scelto un insieme X di risposte da un test istanza di Valutazioni:

- *è* certamente computazionalmente poco costoso verificare se la valutazione delle risposte al test in X distribuisce al più i 100 punti fissati come votazione massima.

  A tal fine, è sufficiente sommare i voti delle risposte al test incluse in X e verificare che il risultato non superi 100;

- invece *può essere* computazionalmente molto costoso certificare che X assegna la migliore votazione possibile per il test in questione, non superando la votazione massima ammissibile.

  Il motivo è che *intuitivamente* non esiste altro modo se non esplorare tutti i possibili sottoinsiemi di domande per stabilire se X offre la migliore votazione.

Ovviamente, analoghe osservazioni valgono per i problemi Bando e Rischio.

1.1.2 Domande rilevanti

- Possiamo sostituire l'*intuizione* con la *certezza* che è inevitabile correre il rischio di dover fare molta "fatica computazionale" nel certificare che X fornisce la votazione migliore, cioè che X è ottimale?

- Supponendo di saper rispondere affermativamente alla prima questione, come si descrive lo spazio in cui cercare X:

  - individuando appena possibile le parti di spazio che con *assoluta certezza* non potranno mai condurre alla certificazione?
  - terminando appena possibile la ricerca di X, con la certezza che X è una delle migliori risposte possibili?

1.2 Discussione post-introduzione

La discussione conclusiva a seguito della introduzione dei problemi motivazionali, riassunta ultra sommariamente in discussione-post-introduzione.pdf, ha immediatamente portato ad inquadrare la ricerca di algoritmi in grado di risolvere i problemi motivazionali nell'ambito della Programmazione Lineare (PL).

Pur essendo corretta l'idea di ricondursi a tecniche algoritmiche note per risolvere problemi, non è possibile immaginare di sfruttare *tout-court* algoritmi forniti nell'ambito della PL per certificare l'ottimalità di una soluzione a Valutazioni o a problemi analoghi.

Aspetto fondamentale di Valutazione, Bando e Rischio è che soluzioni per essi sono costituite da insiemi di elementi che *non ammettono frazionamenti*.

Ad esempio, in Valutazioni dobbiamo esibire un insieme di risposte valutate per ottenere il voto finale. Ciascuna risposta va presa nella sua interezza: o è nell'insieme che concorre a determinare il voto finale, o non lo è.

È come dire di rappresentare la i-esima risposta con una variabile x_i. Il vincolo sulla non frazionabilità degli elementi da inserire in un insieme equivale ad imporre $x_i \in \{0, 1\}$, sotto l'assunzione naturale che se $x_i = 0$, allora la risposta i-esima *non è scelta*, mentre se $x_i = 1$, allora la domanda i-esima *è scelta*.

Analogo discorso vale per Bando o Rischio. In entrambi i casi una richiesta di finanziamento o la si asseconda, o non la si asseconda. Anche in questo caso si costruisce un insieme in cui un oggetto sta o non sta.

Al contrario, soluzioni a problemi di PL ammettono valori frazionari per le variabili x_i che appartengono ad uno spazio tipicamente infinito, come i punti in un'area di un qualche (iper)spazio. Nel caso dei problemi Valutazioni, Bando e Rischio occorre parlare di Programmazione Lineare *Intera* (PLI) delle cui risposte (soluzioni ottimali) la PL può dare stime per eccesso.

La discussione si è alla fine orientata ad individuare il problema dello zaino (*Knapsack* o KP) come problema cui ricondursi per risolvere almeno Valutazioni, citando la tecnica *Branch&Bound* (*Branch&Bound*) per risolverne le istanze.

Siccome vedremo che la complessità computazionale asintotica della tecnica *Branch&Bound* è molto alta è stato suggerita una riflessione su una possibile alternativa.

1.2.1 "Compito"

È immaginabile sperare di cercare euristiche *greedy* efficienti per risolvere almeno Valutazioni?

Parte I

Algoritmica

2.1 Euristiche *greedy* aiutano?

Ricapitolazione introduttiva.mp4.

Tra i problemi Valutazioni, Bando e Rischio quello con la specifica più semplice sembra essere Valutazioni. Per semplicità, quindi, focalizziamo la nostra attenzione su di esso.

Ci chiediamo se un'euristica ragionevole di impronta *greedy* può risolvere il problema Valutazioni, individuando un sottoinsieme delle risposte date al test, che fornisce la votazione migliore non superiore al massimo consentito. Tecnicamente, quell'insieme sarà individuato come *risposta*.

Euristiche-greedy-non-aiutano.mp4, associato al documento Euristiche-greedy-non-aiutano.pdf illustra un paio di euristiche naturali per Valutazioni.

Entrambe costruiscono un insieme di risposte per determinare la votazione finale. Sono basate su due fasi, una di ordinamento, il cui risultato non è necessariamente unico, l'altra di scelta. I criteri di ordinamento sono due:

- elencare le risposte in ordine non crescente di voto assoluto ottenuto;
- elencare le risposte in ordine non decrescente del rapporto tra voto assegnato e voto massimo possibile per quella risposta; a parità di rapporto, si può dare priorità alla risposta col voto assoluto maggiore. È un ordinamento che privilegia le risposte con rendimento migliore.

Indipendentemente dal criterio di ordinamento, il voto finale è la somma dei voti assoluti dati alle risposte, partendo dalle migliori, finché la somma non supera il voto massimo consentito, o finché non si esaurisce l'insieme.

La conclusione del tentativo di usare euristiche *greedy* per trovare una risposta a Valutazioni è che esse non sono sempre in grado di trovare almeno una risposta al problema; del resto ce lo aspettavamo. Inoltre non è troppo complicato escogitare

istanze, anche molto compatte, in cui non solo tali tecniche non forniscono la risposta, ma tra le due specifiche strategie naturali trovate, la migliore risulta essere quella che, tendenzialmente, non si userebbe: è la strategia che, ad ogni passo, tenta di massimizzare il più possibile la votazione, inserendo nell'insieme di risposte date al test quelle con votazione assoluta maggiore.

2.2 Visita *Brute-Force* (BF)

Il nostro focus è rimasto dover trovare un algoritmo che risolve il problema Valutazioni a *basso costo computazionale*.

Da un lato le euristiche *greedy* sono effettivamente a *basso costo computazionale*, ma per ciascuna di quelle investigate abbiamo controesempi al fatto che esse siano sempre in grado di trovare la risposta al problema.

Dall'altro, sappiamo che la natura stessa dei problemi da risolvere non permette di avvalerci della PL (*non* PLI).

Siccome siamo ancora di fronte al problema iniziale rappresentato dalla domanda: "*Come facciamo a produrre la votazione ottimale per una qualsiasi istanza del problema* Valutazioni*?*", il nostro scopo diventa avere uno schema risolutivo generale di problemi analoghi a Valutazioni.

Visita-esaustiva-(BruteForce).mp4, basato su Visita-esaustiva-(BruteForce).pdf, illustra come, avendo a disposizione la possibilità di generare o visitare tutte le permutazioni, o i sottoinsiemi, di una collezione iniziale di *item*, è possibile congetturare come la visita dello *spazio degli stati* di Valutazioni procede per individuare *soluzioni ottimali*, cioè *risposte*.

Quindi, Visita-esaustiva-(BruteForce).mp4 suggerisce di "ridurre" la soluzione di Valutazioni ad una visita esaustiva, o *Brute-Force* (BF), di uno spazio di permutazioni o di sottoinsiemi opportuno. (Usiamo informalmente il termine "ridurre" che, vedremo, ha una valenza tecnica rilevante.) In questo modo certifichiamo che le visite *greedy* date non sono ottimali per l'istanza di Valutazioni considerata: la votazione massima possibile è 8.6, a fronte di un voto massimo pari a 9. Invece la migliore soluzione fornita da una delle due euristiche *greedy* è 8.5.

Infine, Visita-esaustiva-(BruteForce).mp4 suggerisce anche che saper generare tutte le disposizioni con ripetizione di k elementi presi da una collezione di n elementi pone le basi per risolvere il problema Bando, seguendo un'impostazione analoga a quella per Valutazioni;

2.3 Invariante per Permutazioni, Disposizioni, Sottoinsiemi

In base alle osservazioni precedenti è necessario sintetizzare un qualche algoritmo che genera tutte e sole le permutazioni di un *array*.

Miriamo ad una struttura algoritmica che, senza stravolgimenti, potrà essere adattato per generare tutte e sole le disposizioni di ordine k di un *array* e tutti i suoi sottoinsiemi.

Cominceremo col raggiungere il nostro scopo, usando un linguaggio imperativo.

Sottolineiamo l'intenzione di usare un linguaggio imperativo perché, per rafforzare la nostra convinzione a proposito della correttezza dei ragionamenti che svilupperemo, ad un certo punto ci rivolgeremo anche al paradigma di programmazione funzionale: in esso discuteremo e dimostreremo con un livello di dettaglio molto maggiore rispetto allo standard, proprietà di algoritmi che generano permutazioni di liste.

Invariante-permutazioni-sottoinsiemi.mp4 suggerisce una possibile strutturazione di un algoritmo che genera tutte le permutazioni di una collezione di valori in un *array*, descrivendo la situazione generica. Il fine è prepararci a scrivere vari algoritmi in accordo con un singolo modello strutturale per generare disposizioni, sottoinsiemi e permutazioni.

Invariante-permutazioni-sottoinsiemi.mp4 si basa sul documento Invariante-permutazioni-sottoinsiemi.pdf la cui ultima pagina segnala il testo "*The Art of Computer Programming Volume 4A Combinatorial Algorithms Part 1*" di Donald Knuth, disponibile (almeno) su:

- `https://archive.org/details/B-001-001-251/mode/2up`
- `https://epdf.pub/`

ricchissimo di dettagli a proposito di algoritmi combinatori.

2.3.1 Generazione effettiva degli spazi

Implementiamo algoritmi con cui sperimentare la generazione di spazi degli stati caratterizzati da una dimensione che può crescere esponenzialmente in funzione della dimensione dell'input.

Il punto di partenza è lo schema in Visita-esaustiva-(BruteForce).mp4 che genera tutte le permutazioni di una collezione di elementi; con sue minime varianti potremo ottenere:

- tutte le permutazioni con ripetizione della collezione;
- tutte le disposizioni di k elementi presi tra gli n della collezione;
- tutte le disposizioni con ripetizione di k elementi presi tra gli n della collezione, tipicamente indicate come $D_{n,k}$;
- tutti i sottoinsiemi di una collezione.

Lo scopo è impostare una struttura comune iniziale ed essenziale del meccanismo di visita esaustiva che costituisce la base di schemi algoritmici, da modificare in punti specifici, per ottenere visite più sofisticate e, *per quanto possibile*, più efficienti.

Il motivo per impostare una struttura comune è che, considerata la complessità dimensionale degli spazi che vogliamo generare, cioè esplorare, occorre essere ragionevolmente sicuri che gli algoritmi usati sono corretti. Il *testing* per essi non è una strategia di "verifica" molto indicata, dovuta proprio alla dimensione immediatamente intrattabile con input appena significativi.

- 00bruteforcePermutazioni.mp4 comincia ad illustrare il lavoro implementativo con cui sperimentare l'efficacia delle tecniche algoritmiche che, poco per volta, introdurremo. L'impostazione è ultra-minimale: siamo concentrati su strutture algoritmiche le più essenziali possibili.

  Il video in questione presenta un'implementazione dell'algoritmo suggerito da Invariante-permutazioni-sottoinsiemi.pdf, illustrando la classe `combinatoricaBF.PermutazioneTest` e del metodo `combinatoricaBF.Permutazioni.risposte`.

- 00bruteforceSottoinsiemi.mp4 illustra macroscopicamente il funzionamento del metodo `combinatoricaBF.Sottoinsiemi.risposta`, mentre 00bruteforceSottoinsiemi-simulazione.mp4 dal minuto *5:15* circa propone una simulazione passo-passo, a mano, della generazione di sottoinsiemi di $\{x, y, z\}$.

- 00bruteforcePermutazioniConRipetizione.mp4 rimarca le principali differenze tra il codice del metodo `combinatoricaBF.PermutazioniRpt.risposte`, che genera permutazioni con ripetizione, e il metodo `combinatoricaBF.Permutazioni.risposte`, mentre 00bruteforcePermutazioniConRipetizione.mp4 fornisce anche una simulazione, tramite Java Visualizer, della generazione di parte delle permutazioni con ripetizione degli elementi in $\{1, 2\}$.

Infine, 00bruteforce.zip contiene anche i sorgenti illustrati.

2.3.2 "Compito"

Prendere i seguenti punti come proposte di riflessione.

C.1 "Inventare" algoritmi alternativi a quelli presentati per generare permutazioni, anche con ripetizioni, e sottoinsiemi.

C.2 00bruteforceDisposizioni(ConRipetizione).mp4 illustra il codice dei metodi `combinatoricaBF.Disposizioni.risposte` e `combinatoricaBF.DisposizioniRpt.risposte`, che generano disposizioni di k elementi presi tra n, sottolineando le variazioni apportate a `combinatoricaBF.Permutazioni.risposte` e `combinatoricaBF.PermutazioniRpt.risposte`, rispettivamente.

Modificare un algoritmo eventualmente "inventato" al punto **C.1** precedente per generare $D_{n,k}$, cioè tutte le disposizioni di k elementi presi tra n.

C.3 Riflettere su possibili descrizioni dell'invariante che guida la sintesi degli algoritmi Steinhaus–Johnson–Trotter algorithm e Heap's algorithm per la generazione di permutazioni.

C.4 Supponendo di avere implementazioni per gli algoritmi Steinhaus–Johnson–Trotter algorithm e Heap's algorithm, immaginare varianti che generano, ad esempio, permutazioni con ripetizione.

Introduzione-alla-lezione.mp4 illustra a grandi linee il piano odierno.

2.4 Visita dello spazio degli stati di Valutazioni

Una volta impostata la generazione di tutte le permutazioni o di tutti i sottoinsiemi di una collezione di elementi, è possibile adattarli con minime variazioni per produrre tutte le risposte a Valutazioni.

- Valutazione-permutazioni.mp4 illustra tre metodi `ValutazionePer.permutazioni`, `ValutazionePer.soluzioni` e `ValutazionePer.risposta`.

  Il primo è un mini adattamento de `Permutazioni.risposte`, il secondo di `ValutazionePer.permutazioni` ed il terzo di `ValutazionePer.soluzioni`.

  Lo scopo è suggerire una metodologia generale di lavoro: si parte col delineare lo spazio degli stati, per poi filtrare le soluzioni, da cui, attraverso un ulteriore filtro, estrarre la o le risposte.

- Valutazione-sottoinsiemi.mp4 illustra il codice dei tre metodi `ValutazioneSot.permutazioni`, `ValutazioneSot.soluzioni` e `ValutazioneSot.risposta`.

  La struttura e la sua giustificazione sono analoghe a quelle per il video Valutazione-permutazioni.mp4, ma focalizzato sul manipolare lo spazio degli stati come insieme di insiemi.

Valutazione-permutazioni.pdf e Valutazione-sottoinsiemi.pdf riassumono graficamente strutture dati su cui lavorano le implementazioni di algoritmi appena descritte.

Infine, 00bruteforce.zip contiene il codice illustrato.

2.5 Visita dello spazio degli stati di Bando

In analogia con Valutazione, ma adattando lo schema algoritmico BF che genera $D_{n.k}$ con ripetizione, cioè tutte le disposizioni di n elementi e classe k con ripetizioni, visitiamo lo spazio degli stati del problema Bando; la necessità di avere ripetizioni deriva dal fatto che numeriamo a partire da '1', ad esempio, i progetti di ciascuna area: il "progetto n" può esistere nelle aree di intervento 0 e 1, ad esempio.

Bando-permutazioni-con-ripetizione.mp4 illustra tre metodi:

- `BandoDisRep.spazioStati` adattato da `DisposizioniRpt.risposte`;
- `BandoDisRep.soluzioni` adattato da `BandoDisRep.spazioStati`;
- `BandoDisRep.risposta` adattato da `BandoDisRep.soluzioni`.

Anche in questo caso lo scopo è suggerire una metodologia generale di lavoro: si parte col delineare una strutturazione dello spazio degli stati, cui corrisponde un algoritmo di generazione, per poi filtrare le soluzioni da cui, attraverso un ulteriore filtro, estrarre la o le risposte.

Bando-permutazioni-con-ripetizione.pdf è una guida a quanto fatto.

Certificazioni 3

Certificazione-introduzione.mp4 presenta i contenuti di questa breve introduzione.

Sia `Permutazioni.risposte`, sia le sue varianti per generare disposizioni o sottoinsiemi di un *array* sono state giustificata dallo schema Invariante-permutazioni-sottoinsiemi.pdf.

Per esempio, quanto è solida la nostra certezza che `Permutazioni.risposte`, per ogni n, lunghezza dell'*array* in input, generi tutte e sole le $n!$ permutazioni? Certamente non è possibile controllare a vista che 100! permutazioni sono tutte distinte.

In mancanza di una definizione di "permutazione" di riferimento non è possibile *certificare* attraverso dimostrazioni dettagliate che `Permutazioni.risposte` è:

- *corretto*, cioè le permutazioni generate sono $n!$, per un *array* iniziale lungo n;
- *completo*, cioè le permutazioni generate sono tutte distinte.

Tuttavia, anche supponendo di aver fissato una nozione "permutazione" di riferimento, qual è il livello formale al quale ci si può spingere nel dimostrare correttezza e completezza di un algoritmo che genera permutazioni di un insieme di elementi?

Risponderemo alle domande precedenti in maniera relativamente esaustiva, definendo (almeno) un algoritmo *funzionale* dal nome eloquente `permutations`.

Macroscopicamente parlando, svilupperemo `permutations` e certificheremo la sua bontà in accordo col seguente programma:

- fisseremo una descrizione formale di "permutazione" che risulterà essere una relazione di equivalenza `Permutation` tra *liste*. Parliamo di *liste* e non di *array* perché useremo esclusivamente notazioni tipiche della programmazione funzionale pura, in cui non esistono i concetti di assegnazione ed accesso diretto a strutture come *array*.

- produrremo piccole dimostrazioni, tipicamente per induzione, per argomentare solidamente sul fatto che `permutations` è corretto e completo rispetto alla generazione di elementi in `Permutation`.

3.1 1mo Passo. Permutazioni come "oggetti" induttivi

Coq-Permutation-intuizione.mp4 commenta a livello intuitivo la sensatezza della definizione `Permutation` nel Listato 3.1 in cui si formalizza una possibile versione dell'idea che due liste sono una la permutazione dell'altra se contengono esattamente gli stessi elementi, ma, ovviamente, non necessariamente nello stesso ordine.

```
Inductive Permutation : list nat → list nat → Prop :=
| perm_nil: Permutation [] []
| perm_swap x y l  : Permutation (y::x::l) (x::y::l)
| perm_skip x l l' :
  Permutation l l' → Permutation (x::l) (x::l')
| perm_trans l l' l'' :
  Permutation l l' → Permutation l' l'' → Permutation l l''
```

Listato 3.1: Permutazioni come relazione

Il Listato 3.1 caratterizza il concetto "permutazione tra liste" sotto forma di relazione, cioè come sottoinsieme del prodotto cartesiano tra l'insieme di tutte le liste (di naturali) e se stesso.

Non possiamo affermare che il Listato 3.1 definisca correttamente il concetto "permutazione" in senso assoluto; possiamo giustificarne la sensatezza in virtù delle seguenti osservazioni:

- è ovvio che lista vuota `[]` è permutazione (banale) di se stessa (`perm_nil`);
- siccome ogni lista `l` è la permutazione di se stessa, due liste che otteniamo da `l` estendendola una volta con `x` ed `y`, in quest'ordine, ed una volta scambiando l'ordine dei nuovi elementi, sono una la permutazione dell'altra (`perm_swap`);
- se due liste `l` ed `l'` sono una permutazione dell'altra, allora le liste che otteniamo aggiungendo un nuovo elemento `x` ad entrambe sono ancora permutazione l'una dell'altra (`perm_skip`);
- infine `perm_trans` ricorda la regola di transitività della eguaglianza, e la lettura dovrebbe essere ovvia se `l` è permutazione di `l'` e se `l'` è permutazione di `l''`, allora `l` è permutazione di `l''`.

Esempio 3.1.1 Coq-Permutation-esempio.mp4 illustra lo spirito della definizione nel Listato 3.1, usando la definizione `Permutation` per stabilire che `[2;1;3;4]` e `[2;3;4;1]` sono una la permutazione dell'altra. Si tratta verificare che la relazione `Permutation [2;1;3;4] [2;3;4;1]` è vera, cioè è costruibile applicando opportunamente le regole definitorie nel Listato 3.1. Una possibile sequenza di applicazione di tali regole è:

- `Permutation [2;1;3;4] [2;3;4;1]` se `Permutation [2;1;3;4] [2;3;1;4]` e `Permutation [2;3;1;4] [2;3;4;1]`, applicando la regola `perm_trans`.
- `Permutation [2;1;3;4] [2;3;1;4]` se `Permutation [1;3;4] [3;1;4]`, applicando la regola `perm_skip`.
- `Permutation [2;3;1;4] [2;3;4;1]`, se `Permutation [1;4] [4;1]`, applicando due volte la regola `perm_skip`.
- `Permutation [1;3;4] [3;1;4]` è vera grazie all'applicazione del caso base `perm_swap`, il quale non si rifà ad alcun'altra situazione più semplice. Lo stesso vale per `Permutation [1;4] [4;1]`.

Riassumendo `Permutation [2;1;3;4] [2;3;4;1]` è vera perché riusciamo a costruire la relazione tra le due liste applicando un numero finito di volte le regole che ci siamo dati. □

L'analogia tra la relazione `Permutation` e una relazione di eguaglianza, accennata durante la lettura del significato di `perm_trans`, non è casuale. Come la nozione di "eguaglianza", `Permutation` è una relazione di equivalenza, cioè è riflessiva, simmetrica e transitiva.

La transitività vale per definizione, grazie alla clausola `perm_trans`, mentre riflessività e simmetricità vanno dimostrate.

3.1.1 `Permutation` è riflessiva

```
Theorem Permutation_refl:
    ∀ l: list nat, Permutation l l.
```

Listato 3.2: Riflessività di `Permutation`

Il Listato 3.2 enuncia la riflessività di `Permutation`, perché si legge come: "Per ogni lista di naturali `l`, `l` è una permutazione di se stessa, cioè possiamo costruire `Permutation l l`, usando le regole del Listato 3.1."

Dimostrazione de `Theorem Permutation_refl.` Procediamo per induzione sulla struttura di `l`.
Caso base con `l` vuota, cioè `l = []`. La tesi da dimostrare è `Permutation [] []` che vale per definizione grazie a `perm_nil`.
Caso induttivo con `l = h::t`, in cui `t` è una lista. L'ipotesi induttiva ci dice che l'enunciato è vero per `t`, cioè che abbiamo: `Permutation t t`; grazie a `perm_skip`, il fatto che `Permutation t t` è vero permette di affermare che `Permutation h::t h::t` è vero. Abbiamo appena dimostrato che `Permutation l l` è vero siccome `l = h::t`. □

3.1.2 `Permutation` è simmetrica

```
Theorem Permutation_sym: ∀ l l': list nat,
                Permutation l l' → Permutation l' l.
```

Listato 3.3: Simmetricità di `Permutation`

Il Listato 3.3 enuncia la simmetricità di `Permutation`, perché si legge come: "Per ogni coppia di liste di naturali `l` ad `l'`, se `l` e `l'` sono permutazioni l'una dell'altra, cioè possiamo costruire `Permutation l l'`, allora possiamo affermare che `Permutation l' l`, usando le regole del Listato 3.1."

Theorem-`Permutation_sym`-dimostrazione.mp4. Assumiamo che `Permutation l l'` sia vera, come richiesto dal predicato da dimostrare. Questo significa che assumiamo di avere `l'` e `l`, una permutazione dell'altra. Per definizione, questo può succedere nei quattro modi espressi dalle regole `perm_nil`, `perm_swap`, `perm_skip` e `perm_trans` del Listato 3.1. Quindi, dobbiamo considerare quattro casi, sfruttando, per ciascuno di essi, le conseguenze di avere che l'assunzione `Permutation l l'` è vera.

1mo Caso base. L'ipotesi è vera grazie a `perm_nil`. Quindi l'ipotesi è dimostrare `Permutation [] []` con `l = []` e `l' = []`. Chiaramente possiamo scambiare `l = []` e `l' = []`, ottenendo la tesi da dimostrare, cioè `Permutation [] []`, che vale grazie a `perm_nil`.

2do Caso base. L'ipotesi è vera grazie a `perm_swap`. Quindi l'ipotesi è `(Permutation x::y::t y::x::t)`, cioè `(l = x::y::t)` e `(l' = y::x::t)`. La tesi da dimostrare diventa `(Permutation y::x::t x::y::t)` che vale per definizione grazie a `perm_swap`, in cui `x` prende il posto di `y` e vice versa.

1mo caso induttivo. L'ipotesi è vera grazie a `perm_skip`, quindi l'ipotesi assume la forma `(Permutation x::t x::t')`, cioè `(l = x::t)` e `(l' = x::t')`. Quindi, deve valere anche la relazione `(Permutation t t')`, in accordo con l'unica regola definitoria di `Permutation` che possiamo applicare in questo caso. Siccome per induzione vale l'implicazione `(Permutation t t' → Permutation t' t)`, possiamo affermare che anche `(Permutation t' t)` è una relazione vera. Ma da essa, grazie a `perm_skip` otteniamo la tesi `(Permutation x::t' x::t)`.

2do caso induttivo. L'ipotesi è vera grazie a `perm_trans`. In questo caso deve esistere una qualche lista intermedia `l''` tale che `(Permutation l l'')` e `(Permutation l'' l')` sono veri. Siccome, per induzione, valgono le implicazioni `(Permutation l l'' → Permutation l'' l)` e `(Permutation l'' l' → Permutation l' l'')`, possiamo affermare che `(Permutation l' l'')` e `(Permuta-tion l'' l)` sono relazioni vere. Applicando `perm_trans` ad entrambe otteniamo che `(Permutation l' l)` è vero. □.

Theorem-Permutation-sym-dimostrazione.pdf è il documento di riferimento.

Osservazioni 3.1.1

- Il livello di dettaglio nelle dimostrazioni appena sviluppate è piuttosto alto e ripercorre quello ancor più particolareggiato, raggiungibile in un *proof-assistant*. Grossomodo, un *proof-assitant* può esser visto come ambiente di sviluppo in cui si definiscono strutture dati, algoritmi che le manipolano, proprietà degli algoritmi e dimostrazioni che tali proprietà valgono.

- Il Listato 3.1 definisce la struttura dati `Permutation` nel *proof-assitant* Coq; i Listati 3.2 e 3.3 sono enunciati di proprietà relative a `Permutation` sempre in Coq. Le dimostrazioni che abbiamo scritto a mano sono completamente sviluppabili grazie alla disponibilità di *tattiche* per la costruzione di dimostrazioni come illustrato nel sorgente `Permutation.v`, disponibile nel *folder* ALeCO_Permutazioni_in_Coq.zip.

  Note 3.1.1 Ho concluso la lezione con una sessione Sessione-Coq-`Permutation` cui, nelle migliori intenzioni, sarebbe dovuta corrispondere una registrazione; l'ho scordata. Questa dimenticanza, però, è in linea con le intenzioni: non ho l'obiettivo di forzarvi a usare il *proof-assistant* Coq. La mia sola intenzione è di propagandare l'esistenza di tali strumenti, affinché, i possibili interessati, possano farsi vivi per eventuali *stage*. Ci saranno comunque altre (poche) occasioni in cui registrerò una sessione di utilizzo del *proof-assisstant*. Il sorgente Coq su cui abbiamo lavorato è `Permutation.v` e contiene il dettaglio della definizione di `Permutation`, di un esempio di dimostrazione che due liste sono una la permutazione dell'altra, in linea con l'**Esempio 3.1.1**, e la dimostrazione che `Permutation` è riflessiva. □

3.2 2do Passo. Distributore di un elemento tra liste

Descriviamo un algoritmo chiave che useremo per ottenere quello finale `permutations` che genera le permutazioni di una lista.

L'algoritmo chiave è `distribute`.

Presi in input un elemento `e` ed una lista `[`e_1`;`e_2`;`e_3`;`e_4`;...;`e_n`]`, `distribute` genera una lista di liste con `n+1` elementi. Sintetizzare in generale la forma degli `n+1` elementi generati da `distribute e [`e_1`;`e_2`;`e_3`;`e_4`;...;`e_n`]` è inutilmente complesso. Piuttosto, è meglio un esempio per comprendere intuitivamente il comportamento di `distribute`.

Esempio 3.2.1 Coq-distribute-esempio.mp4 illustra il processo di sintesi della lista di liste:

```
[[1;2;3;4];[2;1;3;4];[2;3;1;4];[2;3;4;1]]
```

prodotta da `distribute 1 [2;3;4]` così da facilitare la lettura di `distribute` definito come algoritmo funzionale in Pomona.v, sorgente Coq, contenuto nell'archivio ALeCO_Permutazioni_in_Coq.zip.

1. L'approssimazione peggiore del risultato finale prodotto da `distribute 1 [2;3;4]` è `[[1]]`.

2. Un'approssimazione appena migliore si ottiene immaginando di "togliere" `4` da `[2;3;4]` e di ottenere una lista di liste di due elementi a partire dal risultato precedente `[[1]]`. La lista finale di due elementi è `[[1;4];[4;1]]`: a `[[1;4]]` si accoda la lista che otteniamo distribuendo `4` in testa a tutte le liste che già sono nel risultato. In questo caso il risultato contiene solo `[1]`, quindi accodiamo solo `[[4;1]]`.

3. Un'approssimazione migliore della precedente si ottiene immaginando di "togliere" `3` da `[2;3]` e di ottenere una lista di liste di tre elementi a partire dal risultato precedente `[[1;4];[4;1]]`. La lista di tre elementi è `[[1;3;4];[3;1;4];[3;4;1]]`: a `[[1;3;4]]` si accoda la lista che otteniamo distribuendo `3` in testa a tutte le liste che già sono nel risultato, cioè `[[3;1;4];[3;4;1]]`.

4. Il risultato finale si ottiene immaginando di "togliere" `2` da `[2]` e di ottenere una lista di liste di quattro elementi a partire dal risultato precedente `[[1;3;4];[3;1;4];[3;4;1]]`. La lista di quattro elementi è `[[1;2;3;4];[2;1;3;4];[2;3;1;4];[2;3;4;1]]`: a `[[1;2;3;4]]` si accoda la lista che otteniamo distribuendo `2` in testa a tutte le liste che già sono nel risultato, cioè `[[2;1;3;4];[2;3;1;4];[2;3;4;1]]`.

Coq-distribute-esempio.pdf è il documento di riferimento. □

```
Fixpoint distribute (a:nat) (l:list nat): list (list nat) :=
  match l with
    | [] ⇒ [[a]]
    | h :: t ⇒
        (a::l)::
        -- a in cima all'attuale argomento
        (map (fun t' ⇒ h::t') (distribute a t))
        -- testa h della lista l come primo elemento
        -- in ogni lista del risultato parziale
  end.
```

Listato 3.4: Definizione di `distribute a l`

Il Listato 3.4 definisce `distribute` che è basata sulla funzione `map`, tipica del paradigma di programmazione funzionale.

La funzione `map` potrebbe essere nota per via del suo utilizzo nello schema di programmazione MapReduce.

Esempio 3.2.2 Map_examples-interpretazione.mp4 illustra alcuni semplici esempi di funzionamento della funzione `map` tramite Map_examples.v, sorgente Coq. □

La funzione `distribute` distribuisce il parametro formale `a` come primo elemento sia in `l`, sia in tutte le liste che appartengono alla lista di liste generata da `distribute a t` in cui `t` è la coda di `l`.

```
(e1::[e2;e3;e4])
::(map (fun t ⇒ e2::t)
       ((e1::[e3;e4])
        ::(map (fun t ⇒ e3::t)
               ((e1::[e4])
                ::(map (fun t ⇒ e4::t)
                       (distribute e1 [])))))))
```

Listato 3.5: Unfolding di `distribute e1 [e2;e3;e4]`

Il Listato 3.5 illustra l'*unfolding* di `distribute e1 [e2;e3;e4]`. Si osserva l'annidamento dei richiami ricorsivi sino a `distribute e1 []` che produce `[[e1]]`. A quel punto `map (fun t ⇒ e4::t) [[e1]]` distribuisce `e4` in testa a ogni elemento della lista di liste `[[e1]]`, producendo `[[e4;e1]]` cui viene aggiunta `(e1::[e4])`. Il risultato finale è `[[e1;e4];[e4;e1]]`, e così via.

Più in generale, abbiamo che in `map (fun t ⇒ eh::t) [lk;...;ln]`, `map` applica la funzione `(fun t ⇒ eh::t)` ad ogni elemento di `[lk;...;ln]`, generando `[(eh::lk);...;(eh::ln)]`.

Da questa osservazione, e dall'esempio sviluppato con `(distribute 1 [2;3;4])`, l'intuizione può (correttamente) suggerire che, per un qualche `e` fissato, se la lista `l` data in input a `(distribute e)` è lunga `n` elementi, allora la lista di liste generata da `(distribute a l)` è lunga `n+1` elementi. Formalmente:

```
Lemma distribute_length: ∀ (l: list nat) (a: nat),
    (length (distribute a l)) = 1+(length l).
```

Lemma-distribute_length-prologo-alla-dimostrazione.mp4 e Lemma-distribute-_length-dimostrazione.mp4. Fissiamo la lista `l` in input e procediamo per induzione su di essa. Significa considerare la dimostrazione dell'enunciato nei casi in cui `l = []` (lista vuota) e `l = h::t` (lista composta da una testa `h` ed una coda `t`).

Caso base. Assumendo `l = h::t`, l'enunciato da dimostrare è:

```
(length (distribute a [])) = 1+(length []) = 1+0 = 1.
```

Per definizione, `(distribute a []) = [[a]]`, da cui `(length [[a]]) = 1`. Ci siamo ridotti a chiederci se `1 = 1`, che è ovviamente vero.

Caso induttivo. Assumendo `l = h::t`, l'enunciato da dimostrare è:

```
(length (distribute a h::t)) = 1+(length h::t).
```

Per ipotesi induttiva, abbiamo `(length (distribute a t)) = 1+(length t)`. Per definizione, l'enunciato da dimostrare in questo caso induttivo diventa:

```
(length ((a::(h::t))::(map (fun h => a::h) (distribute a t))))
    = 1+(length h::t) .
```

Siccome è ovvio che `(length x::y) = 1 + (length y)`, per qualsiasi `x` e `y`, l'eguaglianza qui sopra diventa:

```
1+(length (map (fun h => a::h) (distribute a t))) = 2+(length
  t)
```

Osserviamo ora il comportamento di `(map (fun h => a::h) (distribute a t))`. La funzione `map` distribuisce `a` come primo elemento di ogni lista di liste ottenuta da `(distribute a t)`, *non alterando* la lunghezza della lista di liste. Quindi, possiamo affermare che `(length (map (fun h => a::h) (distribute a t))) = (length (distribute a t))`. Sostituendo nell'ultima equivalenza, otteniamo:

```
1+(length (distribute a t)) = 2+(length t) .
```

Applicando l'ipotesi induttiva:

```
1+(1+length t)) = 2+(length t) ,
```

che è un'equivalenza ovviamente vera. Si conclude quindi la dimostrazione di `Lemma distribute_length` nel caso induttivo.

Coq-distribute-length-dimostrazione.pdf è il documento di riferimento.

Commenti su Dimostrazione_a_mano-e-Dimostrazione_in_Coq.mp4 □

3.3 3zo Passo. Generare permutazioni di una lista

Introduciamo `permutations`, algoritmo che, presa una lista, ne genera le permutazioni. Dimostreremo che `permutations` genera tutte le permutazioni di una lista; per ora l'obiettivo è capirne il funzionamento.

```
Fixpoint permutations (l:list nat) : list (list nat) :=
match l with
| [] ⇒ [[]]
| h :: t ⇒ concat_map (distribute h) (permutations t)
end.
```

Listato 3.6: Definizione di `permutations`

Il Listato 3.6 definisce `permutations` come algoritmo ricorsivo. Tramite `distribute h`, `permutations` distribuisce la testa `h` dell'attuale lista `l` in input nelle liste che appartengono al risultato prodotto dalla chiamata ricorsiva `permutations t`: intuitivamente, `permutations t` genera tutte le permutazioni di `t` e `distribute h` ne estende l'insieme, distribuendo `h` su ciascuna di esse.

Esempio 3.3.1 Coq-permutations-esempio.mp4 illustra, a ameno di qualche dettaglio, il processo di sintesi della lista di liste sviluppato da `permutations [1;2;3;4]`:

```
[[1;2;3;4];[2;1;3;4];[2;3;1;4];[2;3;4;1]
;[1;3;2;4];[3;1;2;4];[3;2;1;4];[3;2;4;1] ... [4;3;2;1]]
```

Pur non dettagliando la definizione di `concat_map`, possiamo svolgere la definizione nel Listato 3.6, ottenendo:

```
(concat_map (distribute 1)
(concat_map (distribute 2)
(concat_map (distribute 3)
(concat_map (distribute 4) [[]])))) .
```

1. La computazione procede dalla distribuzione più interna `concat_map (distribute 4) [[]]`. Essa genera una lista di liste, distribuendo `4` in tutte le posizioni nell'unica lista `[]` di `[[]]`. Siccome `[]` ha una sola posizione in cui distribuire `4`, abbiamo:

   ```
   (distribute 4) [] = restituisce [[4]] .
   ```

2. La distribuzione successiva è quindi `concat_map (distribute 3) [[4]]` che si sviluppa come segue:

   ```
   concat_map (distribute 3) [[4]]
       = ((distribute 3) [4]) ++ []
       = ([[3;4];[4;3]]) ++ []
       = [[3;4];[4;3]]
   ```

   Osserviamo che `((distribute 3) [4])`, in base alla descrizione di `distribute` fatta in precedenza, costruisce una lista di due liste che rappresentano le due permutazioni dei due elementi `3` e `4` che diventa l'input del passo successivo.

3. Possiamo quindi interpretare `(distribute 2) [[3;4];[4;3]]`, ottenendo quanto segue:

   ```
   (distribute 2) [[3;4];[4;3]]
     = ((distribute 2) [3;4]) ++ ((distribute 2) [4;3]) ++ []
     = ([[2;3;4];[3;2;4];[3;4;2]]) ++
       ([[2;4;3];[4;2;3];[4;3;2]]) ++ []
     = [[2;3;4];[3;2;4];[3;4;2];[2;4;3];[4;2;3];[4;3;2]]
   ```

4. Seguendo lo stesso meccanismo, nel passo successivo interpretiamo `(distribute 2) [[3;4];[4;3]]`:

   ```
   (distribute 1)
       [[2;3;4];[3;2;4];[3;4;2];[2;4;3];[4;2;3];[4;3;2]]
    = ((distribute 1) [2;3;4]) ++ ((distribute 1) [3;2;4])
      ++ ((distribute 1) [3;4;2]) ++ ((distribute 1) [2;4;3])
      ++ ((distribute 1) [4;2;3]) ++ ((distribute 1) [4;3;2])
      ++ []
    = [[1;2;3;4];[2;1;3;4];...;[4;3;1;2];[4;3;2;1]]
   ```

   in cui la lista di liste finale `[[1;2;3;4];[2;1;3;4];...;[4;3;1;2];[4;3;2;1]]` contiene 24 elementi, come atteso.

Coq-permutations-esempio.pdf è il documento di riferimento. □

3.4 4to Passo: permutations è completo

Ricordiamo che dimostrare la completezza di `permutations` consiste nell'essere certi che essa produce un numero di permutazioni pari al fattoriale della lunghezza della lista di input. Formalmente:

```
Theorem permutations_completeness: ∀ (l: list nat),
    (length (permutations l) = (factorial (length l))) .
```

Theorem-permutations_completeness-dimostrazione.mp4. Fissiamo la lista `l` in input e procediamo per induzione su di essa. Significa considerare la dimostrazione dell'enunciato nei casi in cui `l = []` (lista vuota) e `l = h::t` (lista composta da una testa `h` ed una coda `t`).

Caso base. Assumendo `l = []`, l'enunciato da dimostrare è:

```
(length (permutations []) = (factorial (length []))) .
```

Valgono le seguenti equivalenze, grazie alle definizioni di definizione, `permutations []`, `length` e `factorial`:

```
(length (permutations []) = (factorial (length [])))
        ↔ (length [[]]) = (factorial 0)
        ↔             1 = 1
```

in cui `1 = 1` è ovviamente vera ed il caso base è dimostrato.

Caso induttivo. Assumendo `l = h::t`, l'enunciato da dimostrare è:

```
(length (permutations h::t) = (factorial (length h::t))) .
```

Per ipotesi induttiva, l'enunciato deve valere per la coda `t` di `l`, che è più corta di un elemento, rispetto ad `l`. Quindi, per ipotesi induttiva è vero:

```
(length (permutations t) = (factorial (length t))) .
```

Vediamo come poter sfruttare l'eguaglianza scritta qui sopra:

S.1 È ovvio che `length h::t = 1+(length t)`, cioè che `h::t` ha un elemento in più rispetto a `t`;

S.2 Per definizione, sappiamo che `permutations h::t` calcola `(permutations t)` e che ad *ogni lista* `l'` che appartiene a `(permutations t)` applichiamo `distribute h l'`. Il `Lemma distribute_length` assicura che la lista di liste prodotta da `distribute h l'` contiene `1+(length l')` elementi.

S.3 L'ipotesi induttiva dice che `length (permutations t) = (factorial (length t))`. Combinando l'equazione dell'ipotesi induttiva con la conclusione tratta al passo precedente, possiamo di dire che:

> "Ripetiamo `(distribute h l')` per `(factorial (length t))` volte, ogni volta producendo una lista di `1+(length l')` elementi."

Cioè, `((distribute h) (permutations t))` genera una lista di liste con `(1+(length l'))*(factorial (length t))` elementi.

S.4 Quanto sono lunghe le liste che qui sopra abbiamo identificato come `l'` ed alle quali applichiamo `(distribute h)`? Esse sono prodotte da `permutations t`. *Siccome l'idea è che* `permutations t` *produce permutazioni di* `t`*, la loro lunghezza non può essere altro che* `(length t)`; cioè deve essere vero `(length l') = (length t)`.

S.5 Sostituendo `(length t)` a `(length l')` in `(1+(length l'))*(factorial (length t))` otteniamo che `((distribute h) (permutations t))`, cioè `(prmutations h::t)` genera una lista di liste con `(1+(length t))*(factorial (length t)) = (factorial (length h::t))`, concludendo la dimostrazione del caso induttivo.

Coq-permutations-completezza.pdf è il documento guida. □

Osservazioni 3.4.1 Theorem-`permutations_completeness`-dimostrazione-in-Coq.mp4 illustra alcuni aspetti ignorati dalla dimostrazione sviluppata qui sopra, nonostante essa sia piuttosto dettagliata:

- La ripetizione di cui si parla in **S.3** è un passaggio non ovvio, che corrisponde a dimostrare:

```
Lemma length_concat_map_distribute:
forall (ll:list (list nat)) (n a: nat),
(forall l, In l ll -> length ((distribute a) l) = S n)
-> length (concat_map (distribute a) ll) = (S n)*(
    length ll)
```

  a riga `165` de Pomona.v in ALeCO_Permutazioni_in_Coq.zip, e che si legge come segue:

  > "Presa una qualsiasi lista di liste `ll`, supponiamo che distribuendo un elemento `a` i testa ad ogni lista `l` in `ll` origini una lista di liste lunga `1 + n`, per un qualche valore `n` fissato. Allora `permutations a ll` genera una lista di liste lunga `(1 + n) * (length ll)`."

- **S.4** contiene un'affermazione evidenziata che, a tutti gli effetti, è una *congettura*: noi *immaginiamo* che `(permutations t)` produca permutazioni di `t`; non lo abbiamo certamente dimostrato. Stiamo cioè dicendo che la completezza, in realtà, dipende dalla *correttezza*, la quale può essere formalizzata come:

```
Theorem permutations_correct : ∀ (l l' : list nat),
    In l' (permutations l) → Permutation l l' .
```

Esso afferma che se `l'` è nel risultato generato da `permutations l`, allora `l'` ed `l` sono una la permutazione dell'altra; `Theorem permutations_correct` è dimostrato in tutti i dettagli a linea `109` de Pomona.v in ALeCO_Permutazioni_in_Coq.zip. □

ALeCO_Permutazioni_in_Coq.zip contiene l'intero ambiente per dimostrare correttezza e completezza di `permutation`, usando il *proof-assistant* Coq. In particolare, l'archivio contiene un *folder* di nome `html` il cui file `index.html` è la radice della documentazione.

3.5 Conclusione sulla certificazione

È evidente che `Permutazioni.risposte` non è esattamente la versione imperativa di `permutations`.

Contrariamente a quanto mi ero prefissato, Due-video-in-uno.mp4 tratta due piccoli argomenti che avrei voluto trattare separatamente, per una questione di ordine espositivo. Contiene quanto segue:

- Traccia corrispondenze intuitive tra il funzionamento del metodo `Permutazioni.risposte`, sintetizzato a partire dall'invariante in Invariante-permutazioni-sottoinsiemi.pdf, e la funzione `permutations`. A grandi linee:
  - l'algoritmo iterativo ha un approccio *top-down*. Predispone tante nuove radici, una diversa dall'altra, di sotto-alberi che rappresenteranno permutazioni, quanti sono gli elementi della lista iniziale da permutare. Genera i sotto-alberi, e distribuisce la corrispondente radice su ogni permutazione che il sotto-albero associato genera.
  - l'algoritmo ricorsivo ha un approccio *bottom-up*. Si immagina di saper generare tutte le permutazioni della lista iniziale, privata del suo primo elemento. Quindi, distribuisce l'elemento iniziale in ogni posizione di ogni lista nella lista di liste generata induttivamente.

  Permutazioni-iterativo-vs-ricorsivo.pdf è il documento di riferimento.

- Descrive il funzionamento di `myperm`, funzione definita a riga `152` de MyPerm.v in ALeCO_Permutazioni_in_Coq.zip; `myperm` mima `Permutazioni.risposte` molto più fedelmente di quanto faccia `permutation`, ma per `myperm` manca gran parte del lavoro di certificazione.

  Coq-myperm-intuizione.pdf è il documento di riferimento corretto rispetto a quello commentato in Due-video-in-uno.mp4: l'errore è nato dall'aver usato una `distribute` in `myperm` con un comportamento differente (più semplice) rispetto a quello della omonima funzione `distribute` in `permutations`.

Backtracking 4

4.1 Introduzione

Siamo partiti dal voler risolvere i problemi Valutazioni, Bando e Rischio, definiti motivazionali.

Intuitivamente, tutti e tre i problemi richiedono di individuare almeno un sottoinsieme delle istanze di tuple date come input. Le tuple sono "valutate" secondo un paio di criteri. Tipicamente un criterio richiede di non superare una certa soglia, riguardo ad una misura; l'altro criterio richiede la migliore quantità di una seconda misura, sempre rispettando la soglia sulla prima.

Un'intuizione iniziale, almeno per Valutazioni, potenzialmente il più semplice tra i tre problemi dati, può consistere nell'adottare euristiche *greedy* per cercare una risposta, cioè una soluzione con specifiche caratteristiche di qualità. Ricordiamo che con euristica *greedy* intendiamo una *strategia a basso costo computazionale* — che impiega una quantità di risorse di calcolo, tipicamente il tempo, accettabile — in grado di fornire una risposta, massimizzando l'incremento di una qualche misura ad ogni passo compiuto. La massimizzazione è fatta in base a valutazioni locali legate allo stato raggiunto in un preciso momento di visita dello spazio degli stati.

Il fallimento di euristiche *greedy* naturali è stato evidente anche su istanze essenzialmente banali di Valutazioni. Siccome Bando e Rischio sono molto simili a Valutazioni, è intuitivo che sia possibile costruire istanze per essi sulle quali tecniche *greedy* falliscono nel trovare una soluzione ottima, cioè una risposta.

Una certa dose di pragmatismo è parsa l'opzione obbligata per risolvere i problemi dati. La strategia che abbiamo assunto è scrivere algoritmi in grado di enumerare tutte le soluzioni e, in ultima istanza, almeno una risposta. Tutte le possibili soluzioni si elencano sapendo generare permutazioni, disposizioni, con e senza ripetizioni, o sottoinsiemi di un insieme di elementi dati, che costituiscono l'input del problema da risolvere:

- per Valutazioni è stato evidente poter sfruttare lo spazio degli stati organizzato sia come permutazioni, sia come sottoinsiemi;
- per Bando il punto di partenza proposto è stata la generazione di disposizioni con ripetizione.

Il difetto, in entrambi i casi, è che, nel caso peggiore, occorre ispezionare tutte le soluzioni per scoprire che la riposta è proprio l'ultima di esse.

Siccome il *numero di soluzioni cresce esponenzialmente* con la dimensione dell'istanza del problema, la soluzione della visita esaustiva, o *Brute-Force*, è inaccettabile. Basti pensare alla dimensione dello spazio degli stati generato da ogni istanza con 50 domande di Valutazioni. Lo spazio contiene 50! permutazioni o 2^{50} sottoinsiemi.

4.2 *Backtrack*

È una delle tecnica algoritmiche che migliora la tecnica *Brute-Force* tramite l'introduzione dei concetti:

- *funzione bound* che individua una misura sulla qualità delle soluzioni che potranno essere prodotte da un sotto albero dell'albero degli stati;
- *pruning* (potatura) di ogni sotto-albero per il quale la *funzione bound* predice l'impossibilità di generare soluzioni di qualità migliore rispetto alla migliore trovata sinora.

4.2.1 Introduzione della tecnica algoritmica

Backtracking-motivazione.mp4 illustra un esempio da cui è evidente il motivo per cui è inutile esplorare un intero sotto albero di Valutazioni, non appena, da un certo punto in poi, gli stati non possono più individuare soluzioni.

Backtracking-motivazione.pdf è il documento di riferimento.

4.2.2 Schema generale di un algoritmo *Backtracking*

Backtracking-pseudoalgoritmo.mp4 illustra i passi da seguire per reimpostare lo schema algoritmico *Brute-Force* in uno schema algoritmico per *Backtrack*, in modo da evitare la visita di interi sotto-alberi, in accordo con la funzione criterio (o *bound*).

Backtracking-pseudoalgoritmo.pdf è il documento di riferimento.

4.2.3 Un algoritmo BT per Valutazioni

BT-Valutazione-implementazione.mp4 illustra un esempio concreto di funzione *bound*, con conseguente azione di *pruning*, attraverso l'implementazione `01backtracking.ValutazioniSot` di un algoritmo BT per Valutazioni, ottenuto dalla struttura di pseudo-algoritmo BT, applicando minime variazioni.

La terza pagina de Backtracking-motivazione.pdf è il documento di riferimento.

4.3 *Backtrack* e problemi classici

Per fissare il concetto di *funzione bound* e conseguente azione di *pruning* di parti dello spazio degli stati di un problema attraverso un'algoritmo BT, esploriamo la soluzione di alcuni problemi classici attraverso, appunto, algoritmi di *Backtracking*.

4.3.1 Ordinamento

Backtracking-ordinamento.mp4 illustra la tecnica *Backtracking* come una possibile implementazione dell'ordinamento.

Backtracking-ordinamento.pdf è il documento guida.

Backtracking-ordinamento-implementazione.mp4 commenta `OrdinamentoPer.ordinamento`, metodo che ordina un *array*, implementando lo schema dello pseudo-algoritmo in Backtracking-pseudoalgoritmo.pdf, fornendo primi esempi di metodi `completa`, `rifiuta` e `accetta`, da applicare ad una soluzione o ad un prefisso di soluzione.

4.3.2 Cammini Hamiltoniani

Definizione 4.3.1 Sia dato un grafo $G = (V, E)$.

Un cammino Hamiltoniano in G è una sequenza di archi che permette di visitare ogni nodo di G esattamente una volta.

Lo scopo del problema è stabilire se, dato un qualsiasi G, in esso esiste un cammino Hamiltoniano. □

BT-Cammini-hamiltoniani-intuizione.mp4 ricorda i concetti di Cammino e Circuito Hamiltoniano in un grafo non orientato, il problema di ricerca associato e discute una possibile *funzione bound*.

BT-Cammini-hamiltoniani-intuizione.pdf è il documento di riferimento.

BT-Cammini-hamiltoniani-implementazione.mp4 illustra una possibile implementazione di `CamminoHamiltonianoPerOgni.risposte` di un algoritmo BT che ricerca tutti i Cammini Hamiltoniani in un dato grafo; lo spazio degli stati è organizzato come permutazioni dei nodi del grafo. In realtà, il video commenta anche le semplici variazioni per ottenere una implementazione che interrompe la ricerca al primo Cammino Hamiltoniano individuato.

BT-Cammini-hamiltoniani-implementazione.pdf è il documento di riferimento.

4.3.3 Colorazione di un grafo

Definizione 4.3.2 Siano dati un grafo $G = (V, E)$ ed un insieme di colori C.

Lo scopo del problema è determinare una colorazione di tutti i vertici V di G, usando solo colori in C, in modo che vertici adiacenti abbiano colore diverso. □

BT-Colorazione-grafo-intuizione-e-implementazione.mp4 ha due finalità:

- introduce il problema, anche illustrando un esempio che guida l'intuizione per ricavare la *funzione bound*.

  BT-Colorazione-grafo-intuzione.pdf è il documento di riferimento.

- illustra una possibile implementazione BT. L'implementazione è basata su una *rappresentazione a permutazioni con ripetizione* dello spazio degli stati, cioè dei colori da usare per colorare i nodi del grafo. L'implementazione sfrutta la versione *Backtrack* dell'algoritmo per generare tutte le permutazioni di un insieme di elementi, ma con ripetizione. Questo implica la necessità di usare tanti colori quanti sono i nodi del grafo. Non è una limitazione; è una scelta naturale se si vuole essere certi di avere almeno una colorazione.

  BT-Colorazione-grafo-implementazione.pdf è il documento di riferimento.

4.3.4 Un algoritmo BT per Subsetsum

Definizione 4.3.3 Siano dati un insieme numerico finito X ed un numero S. Determinare l'esistenza di un sottoinsieme Y di X, tale che la somma di tutti gli elementi di Y sia pari ad S.

□

BT-Subsetsum-intuizione.mp4 illustra come risolvere una istanza di Subsetsum.

BT-Subsetsum-sottoinsiemi-intuizione.pdf è il documento di riferimento.

BT-Subsetsum-sottoinsiemi-implementazione.mp4 illustra un possibile algoritmo BT che risolve Subsetsum, basato su uno spazio degli stati organizzato a sottoinsiemi.

BT-Subsetsum-sottoinsiemi-implementazione.pdf è il documento di riferimento.

`01backtracking.zip` è l'archivio di tutti i sorgenti illustrati.

4.3.5 Compito

Eventualmente partendo da `01backtracking.zip`:

- apportare le modifiche necessarie alla classe `ColorazioneGrafoPerRepOgni` per ottenere la colorazione di un grafo, se possibile, usando un numero di colori prefissato, che non sia necessariamente identico al numero di vertici del grafo.
- sviluppare un metodo che risolve Subsetsum, restituendo solo la prima risposta trovata, rappresentando lo spazio degli stati come permutazioni.

4.4 Panoramica riassuntiva sul *Backtrack*

Con una panoramica de [HSR07, Capitolo 7] giustifichiamo quanto sviluppato sinora, commentando terminologia, esempi, analogie e differenze tra [HSR07, Capitolo 7] e quanto fatto da noi.

4.4.1 Un inquadramento del *Backtrack*

Horowitz-Capitolo7-parte-01.mp4, dopo alcuni cenni storici, sottolinea l'ambito generale di applicazione della tecnica algoritmica *Backtrack* ed una sua descrizione astratta in un paio di punti:

1. La struttura dei problemi a cui si applica è descrivibile in termini del prodotto cartesiano $S_1 \times \ldots \times S_n$ di domini di valori $S_1, \ldots, S_n$, tipicamente finiti, e da una *funzione criterio* $P : S_1 \times \ldots \times S_n \to \mathcal{V}$. Lo scopo è trovare una o più istanze $(x_{11}, \ldots, x_{1n}), \ldots, (x_{m1}, \ldots, x_{mn}) \in S_1 \times \ldots \times S_n$ che "*soddisfano*" $P(x_1, \ldots, x_n)$. La nozione "soddisfare" dipende dall'obiettivo dello specifico problema da risolvere. In taluni casi, "soddisfare" può significare: "minimizzare il valore di $P(x_1, \ldots, x_n)$ in $\mathcal{V}$". In altri può significare: "$P(x_1, \ldots, x_n) =$ Sì", avendo $\mathcal{V} = \{\text{Sì}, \text{No}\}$.

2. *Backtrack* è parente stretta della tecnica *Brute-Force* perché deve permettere di generare tutte le n-uple in $S_1 \times \ldots \times S_n$. *Backtrack* si differenzia da *Brute-Force* perché considera una *funzione criterio modificata*, la *funzione bound*, in grado di produrre valori in $\mathcal{V}$ *per ogni porzione* di n-upla $(x_1, \ldots, x_j) \in S_1 \times \ldots \times S_j$.

   Fissata una qualsiasi $(x_1, \ldots, x_j) \in S_1 \times \ldots \times S_j$, se un'estensione $(x_{j+1}, \ldots, x_n) \in S_{j+1} \times \ldots \times S_n$ qualsiasi di $(x_1, \ldots, x_j)$ non può migliorare il valore $P(x_1, \ldots, x_j)$ della *funzione bound*, allora si effettua il *pruning*: si interrompe la generazione di tutte le n-nuple che estendono $(x_1, \ldots, x_j)$.

   Il *pruning* evita un massimo di $\#S_{j+1} * \ldots * \#S_n$ tentativi di scovare una risposta al problema, in cui la notazione $\#S_i$ rappresenta la cardinalità di S_i.

Horowitz-Capitolo7-parte-01.pdf è il documento guida di riferimento.

4.4.2 Vincoli espliciti/impliciti e spazio degli stati

Horowitz-Capitolo7-parte-02.mp4 introduce quanto segue.

Vincoli espliciti. L'insieme di questi vincoli determina lo "*spazio delle soluzioni*", cioè lo spazio in cui cercare effettivamente la o le risposte al problema dato.

Vincoli impliciti. L'insieme di questi vincoli determina la relazione tra le variabili che costituiscono soluzioni, determinando, come conseguenza, le proprietà della funzione criterio.

Problema delle 4Regine. È una versione più maneggevole, ma ugualmente significative, del classico problema 8Regine. Serve come scenario per sperimentare l'applicazione della terminologia introdotta.

Subsetsum. Altro scenario di applicazione della terminologia introdotta, arricchito con osservazioni sulla struttura concreta dell'insieme di soluzioni i cui elementi possono avere diversa struttura: *prefissi di permutazioni*, che forniscono *soluzioni/risposte a lunghezza variabile*, *funzioni caratteristiche di insiemi*, che forniscono *soluzioni/risposte a lunghezza fissa*.

Horowitz-Capitolo7-parte-02.pdf è il documento di riferimento.

4.4.3 Lo spazio degli stati è (ovviamente) un albero

Horowitz-Capitolo7-parte-03.mp4 sottolinea, senza alcuna sorpresa, che lo spazio delle soluzioni, inizialmente visto come sottoinsieme di $S_1 \times \ldots \times S_n$, è modellato in maniera naturale tramite alberi, come, del resto, abbiamo fatto sin dall'inizio per la tecnica *Brute-Force*.

L'argomentazione si appoggia ancora sui due problemi di riferimento 4Regine e Subsetsum, discutendo la rappresentazione delle soluzioni in termini di prefissi di permutazioni o funzioni caratteristiche di insiemi, aggiungendo osservazioni sul fatto che, generando una struttura dati concreta, occorrono scelte sull'ordine di generazione dei nodi.

Horowitz-Capitolo7-parte-03.pdf è il documento di riferimento.

4.4.4 Nomenclatura (standard?) relativa allo spazio degli stati

Horowitz-Capitolo7-parte-04.mp4 completa il quadro dei concetti necessari sia alla descrizione della tecnica *Backtrack*, sia ad esplorare nuove tecniche algoritmiche che vedremo. In particolare:

Spazio degli stati. È una rappresentazione concreta costituita da tutti i cammini dell'albero con cui abbiamo deciso di rappresentare il prodotto cartesiano $S_1 \times \ldots \times S_n$ in cui risolvere un problema computazionale caratterizzato da una funzione criterio/*bound*.

Stato del problema. Lo è ogni nodo dello spazio degli stati.

Spazio delle soluzioni. Contiene stati s del problema, ciascuno in grado di soddisfare la seguente caratteristica: il cammino dalla radice ad s individua una soluzione, cioè una n-upla $(x_1, \ldots, x_j)$, con $j \leq n$, che rispetta i vincoli espliciti del problema stesso.

Dovrebbe essere evidente che se l'albero con cui si individuano le n-uple $(x_1, \ldots, x_j)$ costruisce le funzioni caratteristiche di insiemi, allora solo le foglie dell'albero possono essere soluzioni.

Al contrario, se l'albero individua permutazioni degli elementi in $S_1 \times \ldots \times S_n$, allora è possibile che tutti i nodi dello spazio degli stati siano anche nodi soluzione.

Spazio delle risposte. Contiene stati s del problema, ciascuno in grado di soddisfare la seguente caratteristica: il cammino dalla radice ad s individua una soluzione, che, oltre a soddisfare i vincoli espliciti, soddisfa anche quelli impliciti, catturati dalla funzione criterio; cioè la *funzione bound* non può innescare alcun *pruning* durante la costruzione della n-upla soluzione.

Horowitz-Capitolo7-parte-04.pdf è il documento di riferimento.

4.4.5 Classificazione dei nodi stato e strategia di visita *Backtrack*

Horowitz-Capitolo7-parte-05.mp4 classifica i nodi di uno spazio degli stati in funzione del ruolo che assumono durante la visita, con lo scopo di individuare strategie di visita diverse, a seconda di come i vari ruoli sono assegnati:

- un *dead node* è radice di sotto-alberi non più espandibili perché completi o, ad esempio, perché la funzione *bound* ne impedisce l'ulteriore sviluppo;
- un *live node* è tale perché non tutti i suoi figli sono stati generati. Vale la pena osservare che un *live node* può avere altri *live node* come discendenti;
- tra i *live node* esiste un unico *E-node*. L'*E-node* è il nodo del quale si sta per generare almeno un figlio.

  Senza eccessive sorprese dalla classificazione appena illustrata segue che:

  - la *depth-first* in un dato spazio degli stati si implementa trasferendo la condizione di essere *E-node* dal nodo padre y al singolo nodo z figlio di y non appena z viene generato;
  - la tecnica algoritmica *Backtrack* è una *depth-first* che sfrutta una funzione *bound* per individuare appena possibile dei *dead node*.

La conclusione è poter riassumere concisamente che la tecnica *Backtrack* risulta da una visita *depth-first* associata all'uso di una *funzione bound*, che abbiamo visto essere una generalizzazione della *funzione criterio*, per fare il p*pruning* di sottoalberi visitando i quali è certo che non possiamo ottenere una risposta, cioè una soluzione tra le migliori in assoluto.

Horowitz-Capitolo7-parte-05.pdf è il documento di riferimento.

4.4.6 Versioni di algoritmi *Backtrack*

Horowitz-Capitolo7-parte-06.mp4 confronta versioni ricorsive di pseudo-algoritmi per la tecnica *Backtrack* e ne commenta brevissimamente una iterativa:

- Quella fornita da [HSR07, Capitolo 7], pur essendo ricorsivo, è governato da una iterazione esterna che genera tutti i figli dell'*E-node* disponibile nella cella $x[k-1]$ della tupla x destinata a contenere la risposta. Quindi, per ogni espansione controlla che sia accettabile, per mezzo della funzione *bound* B_k. Nel caso lo sia, se si è di fronte ad una risposta, si accetta, altrimenti si prosegue con la visita, ricorsivamente.

  `https://en.wikipedia.org/wiki/Backtracking` dà una definizione più in linea quella di un algoritmo ricorsivo classico, con casi base ed induttivi che delimitano i punti in cui accettare, rifiutare o proseguire nella ricerca, in linea con quanto impostato sperimentalmente.

  Horowitz-Capitolo7-parte-06.pdf è il documento di riferimento.

- Horowitz-Capitolo7-parte-07.pdf è il documento di riferimento che commenta brevissimamente una versione iterativa di pseudo-algoritmo *Backtrack*.

Horowitz-parte del Capitolo 7.pdf è parte de [HSR07, Capitolo 7] da cui è stata sviluppata la lettura ragionata appena conclusa.

4.4.7 Finezza tecnica (ignorabile)

Horowitz-Capitolo7-parte-08.mp4 è inserito essenzialmente per completezza. Esso insiste sul fatto che la forma dello spazio degli stati generato può dipendere o meno dall'istanza del problema. Se c'è dipendenza, la generazione viene detta dinamica, altrimenti è statica. Un caso di generazione dinamica è quello in cui può essere utile ordinare gli elementi con cui costruire una soluzione in base ad un qualche criterio.

Horowitz-Capitolo7-parte-08.pdf è il documento di riferimento.

4.4.8 *Breadth-search* e *D-search*

Modificando le politiche di generazione dei figli dell'*E-node* e di scelta del successivo *E-node*, si aprono strategie alternative alla visita su cui si basa *Backtrack*, che deriva da *Brute-Force*.

- Horowitz-Cap8-010.mp4 promuove l'idea che aver individuato le nozioni *live node*, *dead node*, *E-node* pone la base per visite alternative e più flessibili della *Depth-first* che, ricordiamo, corrisponde a generare un figlio z dell'attuale *E-node* y, immediatamente trasferendo a z la qualifica di *E-node*, ma mantenendo y tra i *live node* se rimangono suoi figli da generare.

  L'alternativa illustrata si compone di alcuni passi:

  - generare tutti i figli $z_1, \ldots, z_n$ dell'attuale *E-node* y;
  - dichiarare y *dead node*;
  - unire (secondo una qualche politica) $z_1, \ldots, z_n$ all'insieme dei *live node*;
  - scegliere il nuovo *E-node* dall'insieme dei *live node* appena aggiornato.

  Horowitz-Cap8-010.pdf è il documento di riferimento.

- Horowitz-Cap8-020.mp4 illustra due possibili visite rese disponibili dall'aver a disposizione tutti i figli dell'ultimo *E-node* tra i *live node*:

  Breadth-first search. Essa risulta dall'organizzazione a coda dell'insieme dei *live node* o, equivalentemente, in accordo con la *policy* FIFO: i figli dell'ultimo *E-node*, che per comodità chiamiamo x, sono accodati a quelli già inclusi tra i *live node*.

  Quindi, i figli di x potranno diventare a loro volta *E-node* solo dopo che tutti i *live node* già presenti al momento del loro accodamento sono diventati *dead node* dopo essere stati *E-node*.

  D-search. Essa risulta dall'organizzazione a *stack* dell'insieme dei *live node* o, equivalentemente, in accordo con la *policy* LIFO: i figli dell'ultimo *E-node*, che per comodità chiamiamo x, sono impilati a quelli già inclusi tra i *live node*.

Quindi, il primo *E-node* dopo x sarà l'ultimo suo figlio impilato nello *stack* dei *live node*.

Note 4.4.1 Il testo di riferimento non dà spiegazioni sul significato della lettera '*D*' nel nome '*D-search*'.

È ragionevole pensare che essa stia per '*Depth*', o '*Deep*'. La *D-search* spinge la visita in profondità perché lo *status* di *E-node* viene assegnato ad un figlio dell'ultimo *E-node*. Fatta quella scelta, la 'forma' della visita dipende dalla struttura con cui si è deciso di organizzare l'insieme dei *live node*. Con l'accodamento LIFO non si determina in maniera assoluta in che ordine inserire un insieme di nuovi nodi:

- se si impilano nello stesso ordine di generazione, diciamo 'da sinistra verso destra' la visita sarà in *post-ordine*;
- se si impilano in ordine inverso rispetto a quello di generazione, diciamo 'da destra verso sinistra' la visita sarà in *pre-ordine*. □

Horowitz-Cap8-020.pdf è il documento di riferimento.

4.4.9 Compito

Partendo dalla struttura ricorsiva degli algoritmi *Backtrack* visti, progettare il nucleo di algoritmi che permettano almeno la gestione FIFO dell'elenco di *live node*.

5.1 Visite più flessibili

5.2 Introduzione

Da ora in poi l'obiettivo generale è lavorare sulle conseguenze dall'aver considerato una visita dell'albero degli stati in cui ogni espansione di un *E-node* x genera tutti i figli di x, mettendoli tra i *live node*, rendendo x *dead node*, e trasferendo ad un *live node* opportuno lo stato di nuovo *E-node*.

Più volte avremo modo di affermare che, macroscopicamente, il nuovo modo di visitare lo spazio degli stati consiste nello *svincolare* la visita dei *live node* dal meccanismo di generazione dello spazio degli stati stesso.

Al fine di meglio individuare le conseguenze dello svincolo appena menzionato, la letteratura ha sviluppato terminologie da considerare *standard*.

5.3 '*Branch*': visione più unitaria

Horowitz-Cap8-025.mp4 illustra l'inizio de [HSR07, Capitolo 8] secondo il quale '*Branch*' è il termine *standard* per individuare una politica di espansione dell'albero degli stati che genera tutti i figli dell'attuale *E-node* x, inserendoli tutti tra i *live node*, rendendo x *dead node*, e trasferendo ad un *live node* opportuno lo stato di nuovo *E-node*. Conseguenze immediate della politica *Branch* sono:

- una *visita in ampiezza* dello spazio degli stati è un caso specifico di visita che segue dalla gestione FIFO dei *live node*,
- una *visita in profondità* dello spazio degli stati è un caso specifico di visita che segue dalla gestione LIFO dei *live node*.
- le gestioni FIFO e LIFO sono i due estremi di un ventaglio di possibili criteri di scelta dell'*E-node* tra i *live node*. Il criterio che vedremo è detto "*Least-cost*".

Grossomodo, e lo suggerisce anche il nome, scegliamo l'*E-node* tra i *live node* con un criterio *Least-cost* se usiamo una qualche nozione di costo per decidere quale sia l'*E-node* più opportuno cui trasferire lo status di *E-node*.

Lo scopo ovvio è accelerare la scoperta di una risposta tra le soluzioni.

Horowitz-Cap8-025.pdf è il documento di riferimento.

5.3.1 "FIFO +*funzione bound*" non sempre migliore di "LIFO +*funzione bound*"

Horowitz-Cap8-035.mp4 sottolinea l'effettiva necessità di rivolgersi a politiche di scelta dell'*E-node* più flessibili di FIFO o LIFO, sfruttando un esempio basato sulla soluzione de 4Regine.

In particolare, per quanto si possa avere l'impressione che una politica FIFO può implicare un *pruning* di sotto-alberi più efficace di quello offerto da una politica LIFO, perché, intuitivamente, sembra evitare la visita inutile di sotto-alberi molto profondi, l'esempio mostra il contrario:

> Una visita dello spazio degli stati di 4Regine con *live node* organizzati in una struttura LIFO (cioè una *D-search*), ovviamente corredata dalla *funzione bound* opportuna, può essere migliore di una visita i cui *live node* sono organizzati in una struttura FIFO (cioè una *Breadth-search*) in cui si usa la medesima *funzione bound*.

Il motivo intuitivo è che la ricerca virtualmente in parallelo dello spazio degli stati offerta da "FIFO +*bound*" può ignorare l'esistenza della risposta che si trova "un 'solo' passo più in basso".

Horowitz-Cap8-035.pdf è il documento di riferimento.

5.4 *E-node* svincolato dalla visita dello spazio degli stati

Horowitz-Cap8-040.mp4 introduce la situazione generica di un algoritmo adatto a visitare tutto lo spazio degli stati, modellato a permutazioni, in cui la scelta dell'*E-node* non è forzata dalla struttura ad albero che lo spazio degli stati inevitabilmente assume.

Horowitz-Cap8-040.pdf è il documento di riferimento.

5.5 Generazione flessibile dello spazio degli stati

5.5.1 Nuove implementazioni

Da Horowitz-Cap8-040.pdf segue la necessità si aggiornare le implementazioni di algoritmi *Brute-Force* già visti con l'obiettivo di svincolare l'uno dall'altro i due seguenti aspetti:

- la generazione della struttura ad albero dello spazio degli stati, e

- la strategia di visita dello spazio stesso.

Sarà possibile organizzare l'insieme dei *live node* a coda, o a *stack*, ovviamente, così come sarà possibile scegliere il prossimo *E-node*, o un nodo da rifiutare, anche casualmente.

Ristrutturare significa trasformare gli algoritmi ricorsivi visti sinora in equivalenti iterativi che saranno molto simili agli pseudo-algoritmi che si trovano tipicamente su testi che trattano di algoritmi per problemi combinatori, ed, in particolare, sul testo [HSR07] di riferimento.

La differenza più netta tra i nostri schemi di algoritmi e quelli dei testi è che noi insistiamo su un livello uniforme di presentazione: non mescoliamo parti intuitive e parti più legate al tipo di struttura necessaria ad ancorare l'intuizione ad un processo implementabile.

- BB-PermutazioniRicFIFO.mp4 illustra il metodo `PermutazioniRicFIFO.risposte` in cui una struttura concreta naturale per rappresentare i nodi dello spazio degli stati è `ArrayList<ArrIntInt>`, dove `ArrIntInt` è una coppia tale che:
  - la prima componente è istanza di `Integer[]` e mantiene una permutazione di elementi;
  - la seconda componente è istanza di `Integer` e mantiene la lunghezza della permutazione contenuta nella prima componente.

  Ad esempio, il nodo radice dello spazio degli stati è rappresentabile come `{<[1,2,3],0>}` in cui le parentesi graffe più esterne delimitano l'istanza di `ArrayList` che contiene la singola coppia `<[1,2,3],0>` in cui il primo elemento `[1,2,3]` è l'array di elementi con cui generare le permutazioni ed il secondo elemento `0` indica che, sinora, abbiamo costruito una permutazione di `[1,2,3]` lunga `0` elementi.

  BB-PermutazioniRicFIFO.pdf è il documento di riferimento.

- BB-PermutazioniRicFIFO-random.mp4 rende evidente la guadagnata flessibilità nel riorganizzare la costruzione della lista dei *live node* indipendentemente dalla struttura ad albero indotta dalla definizione ricorsiva degli algoritmi dati: è ora infatti possibile scegliere casualmente il prossimo *E-node*, così come può essere causale la decisione di rifiutare o meno l'attuale *E-node*

  Relativamente alle osservazioni fatte nel video non ci sono documenti di riferimento.

5.5.2 Gli algoritmi sono ormai inutilmente ricorsivi

Aver svincolato la visita dei *live node* dalla generazione della struttura ad albero, configura la visita dell'intero spazio degli stati come algoritmo che gestisce la lista dei *live node* in funzione delle politiche di accodamento dei nuovi *live node* e della scelta del prossimo *E-node*:

- BB-PermutazioniIterFIFO.mp4 illustra il metodo `PermutazioniIterFIFO.risposte`. È la versione iterativa di `PermutazioniRicFIFO.risposte` che non ha più ragione di essere usato.

  Non ci sono documenti di riferimento.

- BB-SottoinsiemiIterFIFO.mp4 illustra il metodo `SottoinsiemiIterFIFO.risposte`, versione iterativa de `SottoinsiemiRicFIFO.risposte` che non ha più ragione di essere usato.

  BB-SottoinsiemiIterFIFO.pdf è il documento di riferimento.

02branchandbound.zip contiene i sorgenti illustrati.

5.5.3 Compito

Eventualmente partendo da 02branchandbound.zip:

- Trasformare `PermutazioniRicLIFO.risposte` che è *ricorsivo* e visita uno spazio degli stati organizzato a permutazioni in cui la lista dei *live node* è una struttura LIFO.
- Trasformare `PermutazioniRicRptLIFO.risposte` che è *ricorsivo* e visita uno spazio degli stati organizzato a permutazioni con ripetizioni in cui la lista dei *live node* è una struttura LIFO.
- Trasformare `SottoinsiemiRicLIFO.risposte` che è *ricorsivo* e visita uno spazio degli stati organizzato a sottoinsiemi in cui la lista dei *live node* è una struttura LIFO.

5.6 Visite *Least-cost*

Una volta svincolata la costruzione dell'insieme dei *live node* dalla visita, con conseguente maggiore libertà di scegliere il prossimo *E-node*, è possibile trovare alternative alle politiche di scelta FIFO e LIFO.

"*Least-cost*" è il nome che individua scelte dell'*E-node* più flessibili.

Lo scopo è sintetizzare criteri che stimano il costo per trovare una risposta nel sotto-albero di cui ciascun *live node* è radice; lo status di *E-node* sarà assegnato ad un *live node* il cui costo stimato non è superiore a quello degli altri. Il risultato potenziale visitare lo spazio degli stati in modo che:

- da un lato, si tagliano alberi molto profondi, quando non c'è speranza di ricavare una risposta da essi,
- dall'altro, si procede in profondità quando un sotto-albero promette di contenere una risposta.

Lo strumento chiave per ottenere le visite *Least-cost* è la *funzione costo* che stabilisce la graduatoria *ranking* dei *live node*.

5.6.1 *Ranking* dei *live node*

Horowitz-Cap8-ranking.mp4 discute il concetto "*ranking* di un *live node*" con cui si potrebbe stabilire a quale *live node* vada assegnato il ruolo di *E-node* e ne illustra la potenziale utilità sul problema 4Regine.

Quindi, illustra due semplici criteri di *ranking* che, però, risultano immediatamente inutili: calcolarli equivarrebbe a risolvere il problema dato, con un conseguente costo computazionale potenzialmente molto elevato.

La conclusione è che, per mantenere accettabile il costo computazionale del problema da risolvere, occorre rivolgersi a *criteri di stima* del *ranking*.

Lo strumento per farlo sarà la *funzione costo*.

Horowitz-Cap8-ranking.pdf è il documento di riferimento.

5.6.2 Funzione costo dei *live node*

Horowitz-Cap8-funzione-costo-struttura.mp4 illustra argomenti che conducono alla definizione delle componenti la *funzione costo* $\hat{c}(x)$, in cui x è un *live node*, la quale deve stimare il valore $c(x)$, *ranking* esatto del *live node* x, il cui calcolo equivarrebbe a risolvere il problema, con il conseguente onere computazionale associato.

Definizione 5.6.1 La funzione costo $\hat{c}(x)$ di un *live node* x è:

$$\hat{c}(x) = f(h(x)) + \hat{g}(x) \tag{5.1}$$

in cui:

- $h(x)$ è la parte nota e quantifica, secondo una qualche unità di misura, il lavoro svolto per arrivare all'*live node* x,
- $f(\cdot)$ è una funzione *peso* monotona, ovvero che fornisce un valore proporzionale a quello dell'argomento,
- $\hat{g}(x)$ stima una funzione *non nota* $g(x)$ che esprime il costo effettivo per costruire un cammino dal *live node* x ad una eventuale risposta che il sotto-albero con radice x eventualmente contiene. Vale la pena osservare che l'albero di radice x può non contenere alcuna risposta.

  La funzione $\hat{g}(x)$ *è da trovare e dipende dal problema.* □

Horowitz-Cap8-funzione-costo-struttura.pdf è il documento di riferimento.

5.7 Analisi della *funzione costo*

Supponiamo che l'*E-node* x abbia y come figlio appena inserito tra i *live node*. Dati i valori $\hat{c}(x)$ e $\hat{c}(y)$, è naturale assumere che $\hat{g}(x) > \hat{g}(y)$; la giustificazione è che, se il sotto-albero di radice x contiene un cammino ad una risposta z che attraversa y, allora il lavoro da y a z è inferiore di quello da x a z.

Dall'ipotesi $\hat{g}(x) > \hat{g}(y)$ segue necessariamente che il lavoro svolto a partire dalla radice per arrivare all'attuale *E-node* x deve essere non nullo, cioè deve essere vera la condizione $f(h(x)) \neq 0$; se così non fosse avremmo vanificato l'effetto di poter scegliere l'*E-node* più flessibilmente, rispetto alla tecnica *Backtrack*:

- Horowitz-Cap8-funzione-costo-analisi-parte01.mp4 illustra perché se valesse $f(h(x)) = 0$, allora imporremmo irrimediabilmente una visita *D-search*.
- Horowitz-Cap8-funzione-costo-analisi-parte02.mp4 illustra la conseguenza di assumere $f(h(x)) \neq 0$ una volta assunto che x è l'attuale *E-node* e $\hat{g}(x) > \hat{g}(y)$ con y discendente di x.

  Una volta inclusi i figli $w_1, w_2, \ldots, w_n$ di x tra i *live node* è necessario valutare $\hat{c}(z)$, per ogni *live node* z siccome i valori di $\hat{c}(w_1), \hat{c}(w_2), \ldots, \hat{c}(w_n)$ possono superare quelli relativi a *live node* inseriti nella lista prima di $w_1, w_2, \ldots, w_n$ e, quindi, risultare sconvenienti da seguire rispetto ad altri.

Horowitz-Cap8-funzione-costo-analisi.pdf è il documento di riferimento per entrambi i video.

5.8 Strategia *Least-cost* secondo Horowitz

- Horowitz-Cap8-LeasCost-definizione.mp4 definisce la strategia di visita *Least-cost* nel modo ovvio, cioè come quella che valuta $\hat{c}(x)$, per ogni *live node* x, assegnando l'etichetta *E-node* al *live node* y con $\hat{c}(y)$ non superiore a quella degli altri *live node*.

  Inoltre le visite *Breadth-search* e *D-search* sono casi particolari di *Least-cost*:

  - *Breadth-search* segue dal definire $\hat{c}(x)$ come:

    $$\hat{c}(x) = f(h(x)) + 0 \text{ con } f(h(x)) > 0 \ .$$

    Intuitivamente, questo significa assegnare un costo inferiore ai *live node* più vicini alla radice, "rimandando" la discesa nei sotto-alberi dei *live node* più profondi, perché più costosi.
  - *D-search* segue dal definire $\hat{c}(x)$ come:

    $$\hat{c}(x) = 0 + \hat{g}(x) \text{ con } \hat{g}(x) > 0 \ .$$

    Intuitivamente, questo significa ignorare il lavoro noto per raggiungere ogni *live node*: inoltre, siccome $\hat{g}(x) > \hat{g}(y)$ ogni qualvolta y è figlio di x, è naturale immaginare che la visita prosegue verso i figli dell'ultimo *E-node* perché tendono a costare sempre meno.

  Horowitz-Cap8-LeasCost-definizione.pdf è il documento di riferimento.

- Horowitz-Cap8-pseudoalgoritmo.mp4 commenta la struttura di uno pseudo-algoritmo per *Least-cost* introdotto dal testo con una piccola discussione su come l'autore del testo giustifica la correttezza dello pseudo algoritmo proposto.

Lo scopo principale per presentare tale algoritmo è essere esaustivi nel conoscere i contenuti del testo di riferimento [HSR07] così da poter confrontare lo pseudo-algoritmo proposto con le nostre implementazioni di algoritmi per *Least-cost*.

Horowitz-Cap8-pseudoalgoritmo.pdf è il documento di riferimento.

5.9 Una possibile visita *Least-cost* per Subsetsum

Ricordiamo che un'istanza di Subsetsum è $\langle X = \{X_0, \ldots, X_{n-1}\}, s\rangle$. Data un'istanza, risolvere Subsetsum significa trovare un sottoinsieme S di $\{0, \ldots, n-1\}$ tale che $s = \sum_{k \in S} X_k$, ammesso che S esista.

Subsetsum è un problema rilevante, ad esempio, alla base di possibili sistemi di crittografia o di *job scheduling*. Quindi, diversi ricercatori ne hanno studiato funzioni costo efficaci per la scelta dell'*E-node* e per il *pruning*, sui cui si basano algoritmi *Branch&Bound*.

La proposta di funzione costo per Subsetsum in via di discussione non è certamente comparabile in termini di efficienza con le migliori esistenti.

Essa è utile dal punto di vista didattico: pone le basi per algoritmi *Branch&Bound* che risolvono il problema dello zaino (KP) di cui Subsetsum è un caso particolare.

5.9.1 Una funzione costo per Subsetsum

Sviluppiamo una *funzione costo* $\hat{c}(\cdot)$ per Subsetsum con cui scegliere il prossimo *E-node*.

La *funzione costo* $\hat{c}(\cdot)$ mira ad individuare un intervallo di valori in cui il valore s della somma che stiamo cercando può trovarsi; gli estremi dell'intervallo sono generati a basso costo, usando la parte nota del costo di un *live node* da valutare per decidere se può essere preso come *E-node*.

LC-Subsetsum-funzione-definizione.mp4 introduce la *funzione costo* $\hat{c}(\cdot)$ da applicare ad un nodo dello spazio degli stati.

Per migliorare la lettura del significato della funzione costo, i suoi argomenti sono rappresentati come prefissi (cammini) $x[0 \ldots j)$, che per definizione, contengono i nodi $x[0], \ldots, x[j-1]$ tali che $x[k] \in \{0, 1\}$, per ogni $k \in \{0, \ldots, j-1\}$.

Definizione 5.9.1 (Funzione costo Subsetsum) Essa è $\hat{c}(x[0 \ldots j))$ definita come:

$$\begin{aligned}
\hat{c}(x[0 \ldots j)) &\triangleq f(h(x[0 \ldots j))) + \hat{g}(x[0 \ldots j)) \\
f(h(x[0 \ldots j)) &\triangleq \sum_{k \in \{0,\ldots,j-1\}} (X_k \cdot x[k]) \\
\hat{g}(x[0 \ldots j)) &\triangleq \sum_{j \le k \le split} X_k
\end{aligned}$$

in cui *split* è il più piccolo indice tale che:

$$\sum_{j \le k \le (split-1)} X_k \le s \le \sum_{j \le k \le split} X_k \ .$$

Ovvero, a partire da j, il valore di *split* è il primo indice che rende:

- $\sum_{j \leq k \leq split} X_k$ la *più piccola* approssimazione per eccesso di s,
- $\sum_{j \leq k \leq (split-1)} X_k$ la *più grande* approssimazione per difetto di s,

entrambe non necessariamente diverse dal valore s stesso. □

Note 5.9.1 La **Definizione 5.9.1** diventa effettivamente utilizzabile per implementare una visita *Least-cost* per Subsetsum se immaginiamo di invertirne il segno, e di confrontarne il valore con $-s$. Ovviamente questo funziona se assumiamo che tutti i valori assumendo che tutti i valori dell'insieme X che costituisce l'istanza di Subsetsum siano positivi, situazione che possiamo sempre generare, immaginando di sommare un opportuno *offset* a tutti gli elementi dell'insieme ed alla somma s da ottenere.

Sotto tale ipotesi, se una specifica istanza di *live node* $x^*[0 \dots j)$ è tale che $\hat{c}(x^*[0 \dots j))$ assume valore minimo, allora $\hat{c}(x^*[0 \dots j))$ è la più piccola approssimazione per difetto di $-s$ e $\hat{c}(x^*[0 \dots j))$ può essere preso come prossimo *E-node*; a $\hat{c}(x^*[0 \dots j))$ corrisponde un sotto-albero ancora da visitare che, in linea di principio, ha più possibilità di altri di fornire la somma $-s$, siccome la somma che riesce a raggiungere è "più sovrabbondante". □

LC-Subsetsum-funzione-definizione.pdf è il documento associato.

5.9.2 *Least-cost*, FIFO/LIFO, *pruning* per Subsetsum

Studiamo per gradi l'effetto combinato di applicare una visita *Least-cost*, eventualmente con una tra le politiche FIFO, o LIFO, di accumulo dei *live node*, anche applicando il *pruning* al problema Subsetsum.

- LC-Subsetsum-funzione-simulazione.mp4 insiste sull'idea che i vari aspetti algoritmici che via via introduciamo hanno ciascuno un effetto specifico rispetto alla visita dello spazio degli stati.

  Principalmente, il video è una simulazione manuale dell'applicazione della sola *funzione costo* appena sintetizzata, ignorando meccanismi di *pruning*. Si può osservare la non predicibilità sulla scelta del prossimo *E-node* dovuto alla sola misura *Least-cost* adottata.

  Simultaneamente, è possibile osservare che l'effetto della *funzione costo* è influenzato dalle politiche FIFO, o LIFO, di accumulo dei *live node*: a parità di valore della *funzione costo*, la scelta dell'*E-node* dipende proprio da quale *live node* diventa disponibile, in funzione della politica FIFO o LIFO adottata.

  Globalmente, abbiamo un esempio compatto in cui osservare l'effetto di tre fenomeni sulla visita dell'intero spazio degli stati:

  - la scelta dell'*E-node* da parte della funzione costo con cui si realizza la visita *Least-cost*;
  - l'ordinamento che deriva dall'uso di una delle due politiche ovvie di "accumulo" dei *live node* FIFO e LIFO;
  - l'eliminazione di sotto-alberi grazie al *pruning*, il quale impedisce l'accesso a (potenziali) *live node* su cui valutare la *funzione costo*.

LC-Subsetsum-funzione-simulazione.pdf è il documento associato.

- LC-e-Bound.mp4 segnala infine che l'archivio 02BranchboundSubsetsum.zip contiene una gerarchia di classi il cui scopo è ripercorrere sperimentalmente l'analisi fatta sin qui su Subsetsum: da una classe con la maggiore arbitrarietà possibile nella scelta dell'*E-node*, si può scendere alle classi più vincolanti che includono sia la scelta *Least-cost* dell'*E-node*, assieme ad un meccanismo di *pruning*, in base ad una politica FIFO, o LIFO, di accumulo dei *live node*.

5.9.3 Compito

- LC-Subsetsum-compito.mp4 propone di sperimentare una funzione costo, derivabile da quella precedente, con lo scopo di migliorare la ricerca della soluzione.

  LC-Subsetsum-compito.pdf è il documento di riferimento.

- Ipotizzare misure che possono permettere visite *Least-cost* dello spazio degli stati relativi ai problemi 4Regine, Cammino Hamiltoniano e Colorazione Grafo.

5.10 Il problema Knapsack (KP)

Il *Problema dello zaino* o Knapsack, abbreviato come KP, è una generalizzazione del problema Subsetsum.

Lo scopo finale di questa parte di programma didattico è definire una *funzione costo* con per guidare una visita *Least-cost* dello spazio degli stati di KP, cui associare un criterio di *pruning*, al fine di sintetizzare un algoritmo *Branch&Bound*.

Seguiremo principalmente il testo [KPP04] la cui introduzione fornisce motivazioni disparate a favore dell'importanza di KP, il quale:

- cattura l'essenza del concetto intuitivo di processo decisionale di cui valutare l'efficacia;
- permette di formalizzare il processo decisionale con un insieme minimale di vincoli composti da combinazioni lineari di valori;
- vincola la rappresentazione delle decisioni prese a valori di variabili secondo l'interpretazione intuitiva "*decisione presa*: valore della variabile associata pari ad 1" e "*decisione non presa*: valore della variabile associata pari a 0".

Inoltre, l'introduzione di [KPP04], oltre alla tipica interpretazione associata al problema di riempire uno zaino, massimizzando l'utilità del contenuto, senza sforare la capacità massima dello zaino stesso, contiene una brevissima rassegna di possibili riformulazioni più concrete del KP:

- Problema dell'investimento finanziario;
- Problema del taglio (fisico) di oggetti;
- Problema delle spedizioni;

- Problema della valutazione usato come motivazione iniziale e tratto dall'articolo FeuermanWeiss-Mathematical Programming Model etc.pdf del 1973 che illustra come valutare test descritti in Valutazioni attraverso un algoritmo *Branch&Bound* per il KP.

5.10.1 Panoramica iniziale su KP

Pisinger-KP-010.mp4 inquadra il KP da un punto di vista intuitivo, associandolo a diversi ambiti reali in cui è necessario prendere decisioni valutandone l'efficacia. Ricordiamo la classica definizione formale:

Definizione 5.10.1 (*Knapsack*) Dati i "pesi" $w_1, \ldots, w_n$, i "profitti" $p_1, \ldots, p_n$, ed una "capacità massima" C, cercare un'opportuna istanza $x^* = (x_1^*, \ldots, x_n^*)$ di $x = (x_1, \ldots, x_n)$ nell'insieme in $\{0,1\}^n$ modo da:

$$\text{massimizzare} \sum_{k=1}^{n} p_k x_k \ , \tag{5.2}$$

$$\text{a patto che} \sum_{k=1}^{n} w_k x_k \leq C \ . \tag{5.3}$$

□

La **Definizione 5.10.1** inserisce KP nell'ampia classe dei problemi di *Programmazione Lineare Intera* (PLI).

Pisinger-KP-010.pdf è il documento guida.

5.10.2 Applicazioni basilari di KP

Pisinger-KP-020.mp4 illustra esempi basilari di ambiti concreti per i quali la soluzione a problemi tipici può essere ricondotta alla soluzione del KP:

- ambito finanziario in cui si situa il problema motivazionale Rischio;
- ambito logistico, identificato come Airline cargo businness, che verrà usato anche in seguito per illustrare versioni alternative al KP classico;
- ambito produttivo col quale si accenna al problema Taglio assi in cui si tratta di massimizzare il profitto a seguito della richiesta di tagli standard, richiesti da un insieme di clienti e da applicare ad assi di legno;
- ambito valutativo che ha suggerito l'uso di Valutazioni come problema motivazionale.

  In particolare si accenna alla relazione formale tra definizione generale di KP e Valutazioni. Un'istanza di Valutazioni è una terna $\langle (v_1, \ldots, v_{50}), (m_1, \ldots, m_{50}), 100 \rangle$

per la quale occorre trovare una istanza $x^* = (x_1^*, \ldots, x_{50}^*)$ tale che:

$$\left(\sum_{k=1}^{50} v_k x_k^*\right) \text{ ha valore massimo} \tag{5.4}$$

$$\sum_{k=1}^{50} m_k x_k^* \leq 100 \tag{5.5}$$

$$x_1^*, \ldots, x_n^* \in \{0, 1\} \tag{5.6}$$

$$\sum_{k=1}^{50} m_k > 100 \tag{5.7}$$

come descritto dall'articolo originale FeuermanWeiss-Mathematical Programming Model etc.pdf

Pisinger-KP-020.pdf è il documento di riferimento.

5.10.3 "Esempio" di comportamento di KP

Pisinger-KP-025.mp4 illustra visualmente, quindi a livello intuitivo, e su una specifica istanza di KP, la mancanza di ovvie correlazioni tra l'andamento del profitto e quella dello riempimento dello zaino dovute essenzialmente sia al vincolo di dover considerare come non frazionabile ogni elemento da inserire nello zaino, sia all'ordine con cui man mano si considerano i vari elementi.
Pisinger-KP-025.pdf è il documento di riferimento.

5.10.4 Ulteriori ed ultimi aspetti introduttivi di KP

Pisinger-KP-030.mp4 riassume ultimi aspetti introduttivi che possiamo trovare in [KPP04], ricordando che:

- è tipico ricondurre a quella del KP la soluzione a problemi reali con vincoli ulteriori rispetto alla versione con cui si enuncia la versione classica del problema, così come è tipico trovare che KP può costituire uno dei passi per risolvere problemi combinatori più articolati;
- Subsetsum è un caso particolare, molto essenziale, di KP che, proprio per la sua essenzialità, è al centro dello sviluppo della teoria della complessità computazionale relativa al campo della NP-hardness, cioè di quella parte di algoritmica i cui scopo è capire perché certi problemi sono intrinsecamente difficili (li chiameremo anche intrattabili, indicandone una ulteriore sfumatura) e che, sinora, hanno resistito alla scoperta di algoritmi risolutivi efficienti su ogni loro istanza.

  Sempre in relazione allo studio della complessità computazionale, KP contribuisce a determinare una gerarchia formale tra problemi combinatori che ne caratterizza il livello di difficoltà: KP fa parte dei problemi combinatori più semplici;

- secondo The Stony Brook Algorithm Repository, la richiesta di utilizzo di buoni algoritmi risolutivi per KP è piuttosto popolare.

Pisinger-KP-030.pdf è il documento di riferimento.

5.11 Versioni ed estensioni di KP

Pisinger-KP-040.mp4 (Molto lungo) illustra le seguenti versioni di KP associando ad esse interpretazioni ragionevoli, talvolta nell'ambito che abbiamo chiamato Airline cargo businness, talvolta in qualche altro ambito:

- Subsetsum per il quale, come piccolo esercizio formale, si fornisce una "dimostrazione" che esso è una declinazione di KP;
- Bounded KP caratterizzato da variabili *intere* che però possono assumere valori nel segmento $[0, \ldots, n] \in \mathbb{N}$.

  È possibile dimostrare come ridurre una istanza di Bounded KP ad una di KP.
- d-dimensional KP o Multi-dimensional KP in cui si hanno più di un limite superiore di "capacità" da non superare.

  Il problema motivazionale Rischio, meglio noto come Portfolio selection, è, come illustrato, un esempio di 2-dimensional KP.
- Multiple KP è basato sull'idea che ci sono più zaini da riempire; il problema è stabilire in che zaino deve finire un dato oggetto che, ovviamente, non deve poter essere infilato in due zaini distinti simultaneamente;
- Multiple-choice KP basato sull'idea che a fronte della disponibilità di un solo zaino, abbiamo diversi tipi di item e, per ogni tipo, abbiamo più item di diverso peso. Allora, per ogni tipo, il problema è scegliere un solo item per tipo.

  Il problema motivazionale Bando ne è un esempio proprio perché occorre scegliere un solo progetto per area ed ogni area rappresenta un tipo.

Pisinger-KP-040.pdf è il documento di riferimento.

5.12 Algoritmi *greedy* per KP

Abbiamo introdotto varianti ed estensioni del KP, realizzando che i problemi motivazionali Valutazioni, Bando e Rischio sono declinazioni concrete di tali varianti ed estensioni.

Quindi, fornendo algoritmi efficaci di ricerca di almeno una risposta a KP potremo risolvere efficacemente almeno Valutazioni.

Il nostro obiettivo da qui in poi sarà realizzare algoritmi *Branch&Bound* per KP.

5.13 Algoritmi Greedy e Greedy-split per KP

Miriamo a produrre una prima potenziale risposta per una qualsiasi istanza di KP a basso costo computazionale, cioè in tempo lineare o poco superiore, cioè descrivibile come di ordine $n \log n$, come illustrato nel [KPP04, Capitolo 2].

- Pisinger-Cap2-010.mp4 illustra gli algoritmi Greedy per KP.

  L'assunzione, in verità non strettamente necessaria, per analizzare il risultato di tali algoritmi è ordinare gli elementi da inserire nello zaino in base alla loro *efficienza*, ovvero in base al miglior rapporto $\frac{\text{profitto}}{\text{peso}}$: si massimizza il profitto per unità di peso occupato ad ogni inserimento. Assumere questo ordinamento semplifica la presentazione senza limitarne la generalità:

  - l'algoritmo Greedy inserisce un elemento di peso w_i nello zaino se $\overline{w}+w_i \leq c$, in cui $\overline{w}$ è l'attuale peso raggiunto. In caso contrario passa al tentativo di inserire il successivo elemento di peso w_{i+1}, ammesso che tale ulteriore elemento esista;
  - l'algoritmo Greedy-split inserisce un elemento di peso w_i nello zaino se $\overline{w} + w_i \leq c$. In caso contrario si ferma, cioè *non* tenta di inserire ulteriori elementi, anche se essi potrebbero entrare nello zaino.

    Il *più piccolo* valore i per cui $\overline{w} + w_i > c$ è indicato col nome *split*.

  Pisinger-Cap2-010.pdf è il testo di riferimento.

- Pisinger-Cap2-015.mp4 discute la qualità della risposta di Greedy in relazione alla risposta ottimale e presenta una variante Ext-Greedy.

  - Per Greedy esiste una semplice classe di istanze per le quali la risposta che Greedy stesso propone può essere fatta degradare "indefinitamente": è sufficiente che l'*item* in grado di fornire il profitto maggiore, rispetto agli altri, *non* possa essere inserito perché il suo peso eccede per una piccolissima frazione dello spazio disponibile restante, a causa del fatto che è già stato inserito un *item* con un miglior rapporto $\frac{\text{profitto}}{\texttt{peso}}$.
  - Ext-Greedy estende Greedy, eliminandone il difetto evidenziato: nel caso il profitto p_i di un elemento x_i non inserito nello zaino superi z^{G} fornito da Greedy, allora p_i diventa z^{EG}.

  Pisinger-Cap2-015.pdf è il testo di riferimento.

5.14 Rilassamento lineare LKP per KP

- Pisinger-Cap2-020.mp4 introduce formalmente sia il *rilassamento lineare* LKP di KP, sia l'algoritmo Greedy-LKP che produce la risposta ottimale, relativa al rilassamento lineare.

– Formalmente, il rilassamento lineare LKP di KP consiste nel seguente problema:

$$\begin{aligned} \text{massimizzare} & \sum_{k=1}^{n} p_k x_k \\ & \sum_{k=1}^{n} w_k x_k \leq C \\ & x_1, \ldots, x_n \in [0,1] \end{aligned}$$

in cui $[0,1]$ è l'intervallo dei numeri reali tra 0 ed 1.

– L'algoritmo Greedy-LKP riempie lo zaino senza partizionare alcun elemento finché è possibile, cioè fino all'elemento di indice *split* escluso.
A quel punto colma lo zaino, aggiungendo alla capacità:

$$\hat{w} = \sum_{k=1}^{split-1} w_k$$

sino a quel punto ottenuta, la quantità:

$$C - \hat{w}$$

che, ovviamente, è inferiore a w_{split}; contemporaneamente, al profitto $\hat{p}$ ottenuto sino a quel punto, cioè:

$$\hat{p} = \sum_{k=1}^{split-1} p_k$$

aggiunge il valore:

$$\frac{(C - \hat{w})}{w_{split}} p_{split}$$

ovvero una porzione di profitto p_{split} proporzionale al peso $C - \hat{w}$ ancora disponibile per lo riempimento, ma inversamente proporzionale al peso w_{split} che vorremmo inserire. Si ottengono così il profitto:

$$z^{\mathsf{LP}} = \hat{p} + \frac{(C - \hat{w})}{w_{split}} p_{split}$$

ed il corrispondente vettore risposta:

$$\overline{x}^{\mathsf{LP}} = (\underbrace{1, \ldots, 1}_{split-1}, \frac{C - \hat{w}}{w_{split}}, 0, \ldots, 0)$$

nel quale osserviamo che $\frac{C-\hat{w}}{w_{split}} < 1$.

Pisinger-Cap2-020.pdf è il testo di riferimento.

- Pisinger-Cap2-025.mp4 sviluppa la dimostrazione che Greedy-LKP gode della *greedy optimality property*:

  > La scelta migliore su quale oggetto infilare nello zaino, basata su criteri di valutazione *locali*, assicura la scelta migliore *globale*, cioè conduce ad una risposta (soluzione ottimale) al problema.

  Declinata per il KP, la *greedy optimality property* di Greedy-LKP assicura lo riempimento ottimale dello zaino, massimizzando il profitto e colmando il peso, grazie al fatto che *è possibile prendere anche porzioni* di oggetti da infilare nello zaino.

  Tecnicamente, il profitto z^{LP} fornito da Greedy-LKP, applicato ad un'istanza $\langle (w_1, \ldots, w_n), (p_1, \ldots, p_n), C \rangle$, ed assicurato dal vettore $\overline{x}^{\mathsf{LP}} = (\underbrace{1, \ldots, 1}_{split-1}, \frac{C-\hat{w}}{w_{split}}, 0, \ldots, 0)$, è ottimale.

  Pisinger-Cap2-025.pdf è il testo di riferimento.

- Pisinger-Cap2-030.mp4 illustra il significato della seguente gerarchia di inclusioni:

$$\hat{p} \leq z^{\mathsf{G}} \leq z^* \leq \underbrace{\lfloor z^{\mathsf{LP}} \rfloor}_{U_{\mathsf{LP}}} \leq z^{\mathsf{LP}} \leq \hat{p} + p_{split} \leq z^{\mathsf{G}} + p_{split}$$

  tra i valori del profitto offerti dagli algoritmi Greedy (z^{G}), Greedy-split ($\hat{p}$) e Greedy-LKP (z^{LP}). Lo scopo è dimostrare che è possibile trovare velocemente, cioè a basso costo computazionale (al più $n \log n$), estremi inferiori e superiori di un intervallo ragionevolmente piccolo in cui certamente si trovi il profitto ottimale z^*, fornito dalla soluzione ottimale $\overline{x}^* = (x_1^*, \ldots, x_n^*)$, la cui individuazione, nel caso peggiore, può essere limitata da un costo computazionale pari a 2^n.

  Pisinger-Cap2-030.pdf è il testo di riferimento.

5.15 Approssimazioni certificate di KP

Pisinger-Cap2-035.mp4 illustra un criterio che può certificare la qualità della risposta fornita da algoritmi *greedy*.

Il criterio passa sotto il nome di *Relative Performance Guarantee*, o *garanzia (relativa) di prestazione*. Esso valuta in maniera relativa quanto il guadagno fornito dall'algoritmo approssimi quello ottimale. L'approssimazione è tanto più vicina ad 1 quanto più l'algoritmo fornisce una buona risposta. L'approssimazione è quantificabile in termini formali.

Definizione 5.15.1 (k-*approximation*) *Un algoritmo greedy G fornisce una k-approximation, con $0 \leq k \leq 1$ se, per ogni istanza I del problema P che G risolve, il rapporto tra il profitto $z^{\mathsf{G}}(I)$ assicurato da $\mathsf{G}(I)$ e il profitto ottimale $z_P^*(I)$ è almeno pari a k:*

$$\forall I, \frac{z^{\mathsf{G}}(I)}{z_P^*(I)} \geq k \ .$$

La k-approximation è tight, stretta, se non è possibile migliorarla, cioè se esiste almeno un'istanza T del problema, per cui si ha:

$$\frac{z^{\mathsf{G}}(T)}{z^*_P(T)} = k \ .$$

□

Pisinger-Cap2-035.pdf è il documento di riferimento.

5.16 *Branch&Bound* per KP

- Pisinger-Cap2-040.mp4 illustra che Ext-Greedy fornisce una $\frac{1}{2}$-*approximation* per KP.

  Pisinger-Cap2-040.pdf è il documento di riferimento.

- Pisinger-Cap2-045.mp4 illustra che la $\frac{1}{2}$-*approximation* fornita da Ext-Greedy è *tight*: esiste un'istanza che, asintoticamente, "spinge" il valore del profitto $z^{\text{Ext-Greedy}}$ verso $2z^*$. Quindi, non c'è speranza che Ext-Greedy stia in una classe migliore di $\frac{1}{2}$-*approximation*, ad esempio $\frac{3}{4}$-*approximation*.

  Pisinger-Cap2-045.pdf è il documento di riferimento.

- Pisinger-Cap2-050.mp4 introduce l'algoritmo *greedy* $\mathsf{G}^{\frac{3}{4}}$ che, intuitivamente, abbozza la generazione di permutazioni essenziali degli elementi da mettere nello zaino, per poi sfruttare l'algoritmo Ext-Greedy.

  Pisinger-Cap2-050.pdf è il documento di riferimento.

- Pisinger-Cap2-060.mp4 dimostra che $\mathsf{G}^{\frac{3}{4}}$ fornisce una $\frac{3}{4}$-*approximation* per KP.

  Pisinger-Cap2-060.pdf è il documento di riferimento.

- Pisinger-Cap2-070.mp4 dimostra che la $\frac{3}{4}$-*approximation* per $\mathsf{G}^{\frac{3}{4}}$ è *tight*: esiste un'istanza che, asintoticamente, "spinge" il valore del profitto fornito da $\mathsf{G}^{\frac{3}{4}}$ verso il valore $\frac{3}{4}z^*$.

  Pisinger-Cap2-070.pdf è il documento di riferimento.

In linea col titolo, introduciamo un algoritmo *Branch&Bound* per una data istanza $\langle (p_1, \ldots, p_n), (w_1, \ldots, w_n), C \rangle$ di KP con profitti $p_1, \ldots, p_n$, pesi $w_1, \ldots, w_n$ e capacità massima C.

5.16.1 Funzione costo

BB-KP-Funzione-costo.mp4 introduce una *funzione costo* $\hat{c}(\cdot)$ per KP in accordo con la struttura descritta in [HSR07], integrando aspetti che troviamo in [KPP04]. Ricordiamo che:

$$\hat{c}(x[0..j)) = f(h(x[0..j))) + \hat{g}(x[0..j))$$

in cui $x[0..j)$ rappresenta l'*E-node* secondo la convenzione che ad esso corrisponde l'assegnazione di un valore nell'insieme $\{0, 1\}$ alle variabili $x_0, \ldots, x_{j-1}$; qui riassumiamo le componenti in termini più qualitativi per fissarne il significato:

- la parte nota del costo $f(h(x[0..j)))$ viene fatta coincidere con il profitto fornito da $x[0..j)$, cioè essa è la somma di tutti i profitti degli *item* effettivamente inseriti nello zaino sino a quel punto.

  L'idea è che se $x[0..j)$ conclude un cammino, cioè se esso è $x[0..n)$ ed il cammino concluso è una risposta, cioè se $x[0..n) = \overline{x}^*$, allora $f(h(x[0..n))) = z^*$ e non ci sarebbe più nulla da stimare, cioè $\hat{g}(x[0..n)) = 0$.

- lo scopo della parte $\hat{g}(x[0..j))$, cioè del costo da stimare, è fornire la previsione più ottimistica possibile del profitto che il sotto albero di radice $x[0..j)$, indicato con $T[0..j)$, può ancora offrire.

  Per questo motivo $\hat{g}(x[0..j))$ è strutturata in due parti che forniscono i due valori da sommare per ottenere la stima cercata:

  1. la prima parte fornisce il profitto che sicuramente possiamo ancora ottenere visitando $T[0..j)$. Esso include (sommandoli) i profitti di tutti gli elementi che, secondo l'euristica *greedy*, sono inseribili nello zaino perché con indice nell'intervallo $[j..\ split)$, con j incluso e *split* escluso;
  2. la seconda parte inserisce la porzione che il rilassamento lineare di KP ancora permette, dopo che è stato tenuto conto di tutto quanto è inseribile, come identificato dalla prima parte qui sopra descritta .

     La porzione di profitto mancante è proporzionale alla porzione di peso dell'elemento di indice *split*.

- infine, il quadro è completato dalla stima per difetto, descrivibile in tre parti, di quello che possiamo chiamare *profitto potenziale*, avendo come punto di riferimento un generico nodo $x[0..j)$ che vediamo come radice del sotto-albero $T[0..j)$:

  – una prima parte tiene conto del profitto noto che è pari alla somma di tutti i profitti degli item effettivamente inseriti nello zaino sino a quel punto, ricavabili da $x[0..j)$;

  – una seconda parte fornisce il profitto che sicuramente possiamo ancora ottenere visitando $T[0..j)$ sommando i profitti di tutti gli elementi ancora inseribili nello zaino perché con indice nell'intervallo $[j..\ split)$;

  – una terza parte che include i profitti che un algoritmo Greedy può ancora inserire come conseguenza dell'inclusione degli item il cui indice è oltre il valore *split*.

Note 5.16.1 Noi usiamo il concetto "stimato per eccesso", cui corrisponde il termine *upper bound*, seguendo[KPP04]. Al contrario, [HSR07] individua lo stesso concetto come *lower bound*, cioè *limite inferiore*. Il motivo è che in [HSR07] gli algoritmi *Branch&Bound* sono presentati in relazione all'euristica *Least-cost* che forza a pensare in termini di valori negativi del profitto; tuttavia il legame tra *Branch&Bound* e *Least-cost* non è necessario. □

BB-KP-Funzione-costo.pdf è il documento di riferimento.

5.16.2 Invariante

BB-KP-Invariante.mp4 introduce l'invariante per algoritmi *Branch&Bound* che risolvono KP. A titolo riassuntivo, ricordiamo che:

- il valore ottimale del profitto per una istanza di KP è:

$$z^* = \sum_{k=1}^{n} p_k x_k^*$$

  data la risposta $\overline{x}^* = (x_1^*, \ldots, x_n^*)$.

- l'algoritmo Greedy (in [KPP04]) può essere scelto per fornire il primo *lower bound* $z^{\mathsf{G}}(x[0..0))$ per z^*, a basso costo computazionale, cioè:

$$z^{\mathsf{G}}(x[0..0)) \leq z^* \text{ dove } z^{\mathsf{G}}(x[0..0)) = \left(\sum_{k=0}^{split-1} p_k\right) + \left(\sum_{k=split+1}^{n-1} x_k p_k\right)$$

  per una *opportuna assegnazione* di valori in $\{0, 1\}$ agli elementi in $x[split+1..n)$;

- per ogni nodo $x[0..j)$, Greedy-LKP fornisce $z^{\mathsf{LP}}(x[0..j))$, *upper bound* al profitto che il sotto-albero $T[0..j)$ non ancora esplorato, con radice $x[0..j)$, è in grado di assicurare;

- il costo effettivo $f(h(x[0..j)))$ sino ad un qualsiasi *E-node* $x[0..j)$ è noto.

Qualitativamente, l'invariante con cui caratterizziamo un processo di ricerca ha le seguenti caratteristiche:

- *localmente* è a basso costo computazionale;

- *globalmente* può essere molto computazionalmente costoso;

- *globalmente* è in grado di individuare un intervallo di valori via via migliore entro cui z^* dovrà necessariamente trovarsi; cioè l'invariante, se correttamente implementato, assicura un algoritmo *completo*: se una soluzione ottimale esiste, e sappiamo che esiste perché il dominio di ricerca è finito, allora la trova.

Nel dettaglio, l'invariante è strutturato come segue:

Passo base. Il primo *E-node* è la radice $x[0..0)$ dello spazio degli stati: Greedy$(x[0..0))$ produce un primo valore $z^*(x[0..0))$ cui corrisponderà una prima soluzione al problema.

Può capitare che tale soluzione sia anche la risposta: succede se nessuna altra soluzione la migliora.

Passo induttivo. Supponiamo che valgano le due condizioni seguenti:

- $x[0..j)$ è l'*E-node*, per un qualche valore di j;

- la migliore soluzione sinora trovata dall'algoritmo corrisponde ad un nodo $x[0..r)$, per un qualche valore di r, in grado di assicurare il profitto $z^*(x[0..r))$.

Possiamo individuare le seguenti situazioni:

Rifiutare (nodo "completo"). Avviene in due casi:

- se $f(h(x[0..j))) > C$, allora l'intero sotto-albero $T[0..j)$ va potato perché non in grado di produrre *risposte*, avendo sforato la capacità massima;
- se $f(h(x[0..j))) \leq C$ e $f(h(x[0..j))) + z^{\mathsf{LP}}(x[0..j)) \leq z^*(x[0..r))$, cioè pur stando nello spazio disponibile, il massimo profitto che stimiamo di poter ottenere visitando il sotto-albero $T[0..j)$ non può superare la migliore soluzione sinora trovata in corrispondenza di $x[0..r)$. Possiamo quindi potare $T[0..j)$ dichiarandolo completo, perché, con assoluta certezza, $T[0..j)$ non è in grado di produrre alcuna risposta. Osserviamo che la completezza dell'algoritmo è preservata siccome non si esclude alcuna risposta che in $T[0..j)$ non può esistere.

Aggiornare. Se $f(h(x[0..j))) \leq C$ e $z^*(x[0..r)) < f(h(x[0..j)))$, cioè, stando nello spazio disponibile, il profitto $f(h(x[0..j)))$ dell'*E-node* $x[0..j)$ migliora il miglior profitto $z^*(x[0..r))$ sinora noto, allora il miglior profitto deve essere aggiornato a quello fornito da $x[0..j)$ e diventare $z^*(x[0..j))$;

Espandere. Se $f(h(x[0..j))) \leq C$ e $f(h(x[0..j))) + z^{\mathsf{LP}}(x[0..j)) > z^*(x[0..r))$, cioè, stando nello spazio disponibile, l'intero sotto-albero $T[0..j)$ *è ancora in grado* di produrre almeno una risposta, allora occorre continuare ad esplorarlo, espandendo l'*E-node* $x[0..j)$.

BB-KP-Invariante.pdf è il documento di riferimento.

Note 5.16.2 Abbiamo usato il concetto "*lower bound*" coerentemente con [KPP04]. Al contrario, in [HSR07], lo stesso concetto è individuato come *upper bound* per il legame già ricordato tra *Branch&Bound* ed euristica *Least-cost*. □

5.16.3 Implementazioni e risultati

L'archivio `02branchbound.zip` contiene i sorgenti cui si riferiscono i vari punti seguenti.

- KPFIFOBB-Algoritmo.mp4 descrive il metodo `KPFIFOBB.risposte` che implementa un algoritmo *Branch&Bound* per KP con politica FIFO di accodamento dei *live node*.

  Una caratteristica di `KPFIFOBB.risposte` è la leggera ristrutturazione del corpo dell'iterazione principale, rispetto agli algoritmi da noi implementati sinora.

  La riorganizzazione indolore sottolinea quanto sia valida una strategia implementativa graduale attraverso strutture algoritmiche modulari, che sia individuano specifiche proprietà dell'*E-node*, sia sono flessibilmente riorganizzabili.

Sempre KPFIFOBB-Algoritmo.mp4 illustra anche il comportamento di `KPFIFOBB.risposte` sia sulla mini istanza $((2,5,4,2),(2.0,4.6,3.9,2.0),9)$ di Valutazioni, attraverso il metodo `KPLCBBTest.testValutazioniMini`, sia sulla istanza iniziale di Valutazioni; su quest'ultima il metodo `KPFIFOBB.risposte` si rivela estremamente efficace: esso pota essenzialmente l'intero spazio degli stati il quale non è in grado di fornire alcuna risposte migliore di quella fornita inizialmente da dalla visita *greedy*.

- KPFIFOBB-Simulazione-confronti.mp4 è una simulazione a mano della visita dello spazio degli stati effettuata dal metodo `KPFIFOBB.risposte` applicato alla istanza $((2,4,6,9),(10,10,12,18),15)$, comparando la struttura dell'albero prodotto con quella fornita nello **Esercizio 8.3** de [HSR07].

  KPFIFOBB-Simulazione-confronti.pdf è il documento di riferimento.

- KPLCBB-Algoritmo.mp4 descrive il metodo `KPLCBB.risposte` che implementa un algoritmo *Branch&Bound* per KP con politica *Least-cost* di accodamento dei *live node*.

  Sempre KPLCBB-Algoritmo.mp4 illustra anche il comportamento di `KPLCBB.risposte` sia sulla mini istanza $((2,5,4,2),(2.0,4.6,3.9,2.0),9)$ di Valutazioni, attraverso il metodo `KPLCBBTest.testValutazioniMini`, sia sulla istanza fornita dall'**Esercizio 8.2** di [HSR07], corrispondente all'**Example 8.3**, attraverso il metodo `KPLCBBTest.testHorowitzEx8punto2`.

  L'impressione sperimentale conferma l'idea che la politica *Least-cost* possa convergere più velocemente di quella FIFO alla risposta.

- KPLCBB-Simulazione-confronti.mp4 è una simulazione a mano della visita dello spazio degli stati effettuata dal metodo `KPLCBB.risposte` applicato alla istanza $((2,4,6,9),(10,10,12,18),15)$, comparando la struttura dell'albero prodotto con quella fornita nello **Esercizio 8.2** de [HSR07].

  Sempre KPLCBB-Simulazione-confronti.mp4 si conclude con un riassunto tra le notazioni che non sono uniformi tra i testi di riferimento usati.

  KPLCBB-Simulazione-confronti.pdf è il documento di riferimento.

5.17 Osservazioni finali su *Branch&Bound*

Quanto discusso sulla soluzione *Branch&Bound* a KP risulta dal comporre le parti rilevanti dalle Sezione 8.2 in [HSR07] Sezione 2.4 in [KPP04].

Da un lato, l'impostazione del testo [HSR07] pone un forte accento sugli aspetti che servono per tentare di convergere il più velocemente possibile verso una risposta, tramite l'individuazione dell'*E-node* tra i *live node*. È il motivo per cui illustra le politiche *Least-cost* di scelta dell'*E-node*, di cui FIFO e LIFO sono casi particolari. Le tecniche sono di valenza generale, e non necessariamente confezionate per risolvere il KP.

Dall'altro, è chiaro che il testo [KPP04] è focalizzato sulla soluzione del singolo problema KP e non sviluppa accenni particolari alle politiche di gestione dei *live node*

Programmazione dinamica per KP 6

Il riferimento principale per questo argomento è [KPP04, Sezione 2.3].

Intuitivamente, il termine "Programmazione Dinamica" potrebbe essere rimpiazzato da "Pianificazione Dinamica", "Previsione Dinamica"; l'originale "Programmazione" ha un'accezione diversa da quella tipicamente informatica. L'aggettivo "dinamica" ricorda l'idea di un processo che cerca per tentativi la risposta ad una richiesta di pianificazione in cui occorra soddisfare un criterio di ottimalità.

La Programmazione Dinamica è un approccio all'ottimizzazione in cui, molto ad alto livello, si combinano soluzioni ottimali di sotto-problemi in modo da avere una soluzione ottimale del problema che essi compongono.

La [KPP04, Sezione 2.3] suggerisce di vedere l'applicazione della tecnica "Programmazione Dinamica" alla soluzione del KP come sequenza di soluzioni ad un insieme di KP più semplici di quello originale, grazie al controllo delle due dimensioni che caratterizzano il KP stesso, cioè il numero di elementi n, e la capacità massima C.

In particolare, data una istanza di KP con n elementi, e capacità massima C, si può immaginare di produrre una risposta, producendo la risposta al così detto *all-capacities* KP. Ovvero, si tratta di produrre le risposte a tutte le seguenti istanze $\mathsf{KP}_j(d)$ di KP:

$$\begin{aligned} &\text{massimizzare} \sum_{i=1}^{j} p_i x_i \\ &\text{soddosfacendo} \sum_{i=1}^{j} w_i x_i \leq d \ , \end{aligned}$$

per ogni insieme contenente $1 \leq j \leq n$ *item*, per ogni capacità massima $1 \leq d \leq C$.

La "riduzione" della soluzione di una istanza di KP a quella dell'*all-capacities* KP deriva da una rilettura opportuna del processo che può portare alla risposta di una qualsiasi istanza di KP.

6.1 Premessa per una "lettura" ricorsiva di KP

Qui ricostruiamo l'idea che sta alla base del riformulare un'istanza $((w_1, \ldots, w_n), (p_1, \ldots, p_n), C)$ di KP come istanza dell'*all-capacities* KP.

DP-Da-ricorsione-ad-iterazione.mp4, tramite un esempio ricorda come sia possibile trasformare una funzione ricorsiva in una equivalente iterativa.

DP-Da-ricorsione-ad-iterazione.pdf è il documento di riferimento.

6.2 Una prima lettura ricorsiva di KP

DP-Algoritmo-ricorsivo-per-KP.mp4 descrive un processo ricorsivo per risolvere una qualsiasi istanza di KP da cui risulta evidente che si risolve l'*all-instances* KP.

DP-Algoritmo-ricorsivo-per-KP.pdf è il documento di riferimento.

6.3 Una seconda lettura ricorsiva di KP

DP-Algoritmo-ricorsivo-per-KP-seconda-lettura.mp4 approfondisce la descrizione del processo ricorsivo per risolvere una qualsiasi istanza di KP. Da esso risulta evidente che l'albero definito ha dei sotto-alberi condivisi che corrispondono ad istanze dell'*all-instances* KP riutilizzabili, ricordando quanto succede con lo spazio degli stati del processo che calcola i valori della Serie di Fibonacci.

DP-Algoritmo-ricorsivo-per-KP-seconda-lettura.pdf è il documento di riferimento.

6.4 Dalla lettura ricorsiva di KP ad una iterativa

DP-prima-versione-iterativa-di-KP.mp4 illustra come dalla versione ricorsiva di KP, sintetizzata dai punti precedenti, sia naturale sintetizzare almeno una versione iterativa.

È significativo che l'algoritmo iterativo produca in maniera naturale i valori $z_j^*(d)$ di ciascun KP con $j \leq n$ elementi e dimensione massima $d \leq C$ che concorre alla soluzione dell'*all-capacities* KP.

DP-prima-versione-iterativa-di-KP.pdf è il documento di riferimento.

6.5 Ricostruzione della risposta per KP

DP-KP-e-ricostruzione-della-risposta.mp4 osserva che le dipendenze sviluppate all'interno dell'*array* con i valori $z_j^*(d)$ sono sufficienti per ricostruire la risposta, cioè l'insieme di elementi "infilati nell zaino" che massimizzano il profitto.

In tal caso, è possibile ridurre lo spazio utilizzato ad un valore che appartiene a $O(n + C)$ pur rimanendo invariata la complessità in tempo, pari a $O(n * C)$.

DP-KP-e-ricostruzione-della-risposta.pdf è il documento di riferimento.

6.6 Ulteriori versioni iterative di KP

DP-Ulteriori-versioni-iterative-di-KP.mp4 commenta come sia possibile ridurre lo spazio utilizzato, ma non il tempo, osservando alcune specifiche caratteristiche che derivano dalla visita della tabella con in valori $z_j^*(d)$.

DP-Ulteriori-versioni-iterative-di-KP.pdf è il documento di riferimento.

6.7 Complessità di KP in Programmazione Dinamica

DP-KP-e-polinomialita.mp4 illustra il motivo per cui la soluzione in forma di Programmazione Dinamica di una istanza di KP non è polinomiale nella dimensione dell'input; essa è invece *Pseudo-polinomiale* cioè polinomiale nel valore di (almeno) un input.

DP-KP-e-polinomialita.pdf è il documento di riferimento.

Parte II

Intermezzo

7.1 Breve retrospettiva

Breve-retrospettiva-su-quanto fatto.mp4 "legge", ampliandolo un po', quanto scritto qui di seguito.

Abbiamo sin qui seguito un percorso a cavallo tra lo sperimentale ed il metodologico, simultaneamente studiando alcune parti dei testi [HSR07] e [KPP04], in certi frangenti dissezionandoli, perché non così immediatamente trasparenti e formali, "distillando" schemi algoritmici che si sono evoluti nella realizzazione di un nucleo di programmi con cui trovare una risposta ad ogni istanza di KP, problema rilevante scientificamente, quindi, anche dal punto di vista pratico.

Possiamo riassumere le principali tappe:

- Abbiamo stabilito criteri comuni su come visitare esaustivamente spazi di stati rappresentati come permutazioni, disposizioni o sottoinsiemi di elementi, realizzando implementazioni ragionevolmente condivisibili della visita *Brute-Force*.

  Lo scopo principale era avere pochi schemi fissi da usare e riusare senza troppe difficoltà per proporre riposte a problemi intrinsecamente complessi presi come riferimento dalla letteratura: 4Regine, Colorazione di Grafi, Cammino Hamiltoniano, Subsetsum.

  Voler risolvere un problema di riferimento per ricondurvi la soluzione di altri problemi sarà un aspetto rilevante di quanto seguirà.

- Dalla tecnica *Brute-Force* siamo passati alla tecnica *Backtrack*. Il *Backtrack* nasce come *visita in profondità* in cui la discesa lungo un ramo dello spazio degli stati è interrotta quando il nodo visitato non soddisfa criteri opportuni; con *pruning*, o *potatura*, si indica l'interruzione della discesa.

  In accordo con la terminologia definita in [HSR07], il *Backtrack* è una visita dello spazio degli stati in cui, non appena si genera un figlio dell'*E-node*, cioè dell'attuale *live node* da espandere, a quel figlio viene immediatamente

assegnato lo stato di *E-node*, cioè di *live node* specifico; il penultimo *E-node* rimane *live node* finché rimangono suoi figli da espandere.

Il testo [HSR07] indica *Backtrack* come euristica generale ed efficace, almeno in prima battuta, per evitare una visita *Brute-Force tout court*. Quindi, *Backtrack* è la base per euristiche più sofisticate che possono raffinare e sviluppare l'uso della condizione di *pruning*.

- La chiave per applicare il *pruning* in maniera più flessibile e mirata di quanto è possibile tramite il *Backtrack* consiste nello svincolare il procedere della visita di uno spazio degli stati dalla struttura dello spazio stesso.

  Per (ri)dirla come in [HSR07], si tratta di espandere completamente la lista dei figli di un *E-node*, inserendoli tutti immediatamente tra i *live node*. Tra questi si sceglie il successivo *E-node* con criteri che traggono vantaggio dall'aumentata flessibilità.

  I criteri di scelta del prossimo *E-node* più ovvi derivano dal gestire i *live node* con politiche FIFO o LIFO.

  Un criterio più sofisticato è la politica *Least-cost*: il *live node* con valore minimo di un'opportuna funzione costo $\hat{c}(\cdot)$ diventa il prossimo *E-node*. In subordine, a parità di valore, si sceglie l'*E-node* coerentemente con una politica di gestione FIFO o LIFO dei *live node*, per esempio.

- I criteri per definire una *funzione costo* $\hat{c}(\cdot)$ dipendono dai vincoli impliciti (vedi [HSR07]) del problema da risolvere.

  Ci siamo concentrati sulla sintesi di una funzione costo $\hat{c}(\cdot)$ per KP attraverso due punti chiave che, usando la notazione in [KPP04], sono:

  - individuare algoritmi veloci che calcolano approssimazioni per difetto z^{ℓ} della soluzione ottimale z^{*}, per *ogni* istanza di KP;
  - individuare il rilassamento lineare LKP di KP per cui esiste un algoritmo veloce che calcola un'approssimazione per eccesso z^{LP} di z^{*}, per *ogni* istanza di KP.

- Avendo $\hat{c}(\cdot)$, risolviamo KP con la tecnica algoritmica *Branch&Bound*. Ad ogni passo della visita di uno spazio degli stati T, il *Branch&Bound*:

  - usa il miglior valore z^{ℓ} noto per potare ogni sotto-albero t di T la cui migliore approssimazione per eccesso della soluzione che t è in grado di fornire non può superare z^{ℓ};
  - appena possibile, aggiorna z^{ℓ}, visitando i rami non (ancora) potati.

  Inoltre, $\hat{c}(\cdot)$ può essere usata per valutare quanto un sotto-albero t sia "promettente" nel poter produrre una soluzione migliore di quella che ha fornito l'attuale valore z^{ℓ}, risolvendo KP con un algoritmo *Branch&Bound Least-cost*.

- La sintesi di algoritmi *Branch&Bound Least-cost* per KP non risolve il problema del poter incappare in istanze di KP su cui tali algoritmi si comportano in maniera essenzialmente equivalente ad una visita *Brute-Force*: senza alcun

guadagno rispetto al tempo peggiore di visita dello spazio degli stati, cioè esponenziale nella dimensione dell'input.

La Programmazione Dinamica offre un'alternativa alla sintesi di algoritmi per risolvere KP. Il principio è formulare la soluzione di KP in termini ricorsivi affinché la risposta ad un'istanza di KP risulti dall'uso corretto di risposte a istanze più piccole di KP, ottenuto da quella iniziale. Il comportamento della tecnica si riassume dicendo che essa risolve l'*all-capacities* KP.

È interessante che l'algorimto per risolvere una istanza del *all-capacities* KP ha una naturale formulazione iterativa la cui complessità in tempo è *pseudo-polinomiale*.

7.1.1 Parti originali dei Capitoli di riferimento

- Pisinger-Cap1.pdf parte de [KPP04, Capitolo 1];
- Pisinger-Cap2.pdf parte de [KPP04, Capitolo 2];
- Horowitz-Cap7.pdf parte de [HSR07, Capitolo 7].
- Horowitz-Cap8.pdf parte de [HSR07, Capitolo 8].

7.2 Prospettiva futura

Nella prima parte del corso abbiamo visto che non è difficile trovarsi a risolvere problemi definibili come "reali", quali Valutazioni, Bando e Rischio, che risultano essere declinazioni di KP, problema computazionale di riferimento per cui non conosciamo buoni algoritmi che risolvono qualsiasi sua istanza in tempi accettabili, e per cui nessuno è riuscito a dimostrare la non esistenza di tali buoni algoritmi; per questi motivi KP è classificato come problema *intrattabile*, che possiamo interpretare come sinonimo di "intrinsecamente difficile".

Di fronte ad un problema che ci potrà offrire la realtà occorrerebbe avere il "fiuto" di riconoscere se ha caratteristiche che lo rendono intrattabile. Per un problema di quel tipo potremmo anche escogitare un buon algoritmo su istanze che riteniamo tipiche dell'ambito in cui l'algoritmo è impiegato. Dovremo però sapere che la realtà è molto fantasiosa: i "cigni neri" esistono [Tal09]; ne stiamo vivendo una serie in questo periodo (2020, 2021, 2022, ... ?). Prima o poi, un'istanza "cattiva" si presenta all'algoritmo che ha sempre funzionato bene e che, invece, proprio su quella istanza, rivela improvvisamente tempistiche inaccettabili produzione del risultato.

7.2.1 Richiamo su "Riduzione tra problemi"

Lo strumento per capire se un problema B è intrattabile ha natura tecnica. Sapendo che A è un altro problema che già sappiamo essere intrattabile, si tratta di "*ridurre* A a B".

Intuitivamente, "ridurre A a B" significa mostrare che il problema A è un sotto-problema di B. Se sappiamo che A è intrattabile, cioè se nessuno sa come fornirne un algoritmo che si comporti efficientemente su tutte le sue istanze, allora anche B

lo è. Se B è Valutazioni, Bandi o Rischio, allora difficilmente riusciremo ad avere un algoritmo che risolve efficientemente una qualsiasi loro istanza. Il motivo è che essi sono declinazioni reali di KP, il quale, a sua volta, contiene un problema intrattabile da risolvere in piena generalità.

7.2.2 Ripasso di concetti base

Il [GJ79, Capitolo 1] in GareyJohnson-Cap1.pdf costituisce il testo di riferimento per rispolverare concetti e convenzioni utili:

- un video GareyJohnson-Cap1-parte01.mp4 avrebbe dovuto riassume brevemente i contenuti chiave de [GJ79, Sezione 1.1], ovvero uno scenario ipotetico e semiserio che giustifica la teoria della NP-*completeness*, ma non è stato registrato per errore;

- GareyJohnson-Cap1-parte02.mp4 illustra [GJ79, Sezione 1.2] centrata principalmente sul significato delle parole chiave *problema/specifica*, *istanza*, *algoritmi*. In realtà sembra anche fissare convenzioni sulle nozioni di *schema di codifica*, legata al concetto di *istanza*, da cui dipende la *complessità in tempo* per risolvere un problema.

- GareyJohnson-Cap1-parte03.mp4 commenta una prima parte della [GJ79, Sezione 1.3]. Esso inquadra i concetti *schema di codifica* e *complessità in tempo* nell'ambito più ampio rappresentato dalle nozioni di *complessità computazionale* e *problema intrattabile*. Discute sia l'essenziale irrilevanza dell'aumento delle prestazioni computazionali, se posti di fronte a istanze di problemi con complessità esponenziale, sia l'importanza di essere pragmatici: per quanto intrattabili, esistono problemi studiati così a fondo che ammettono tecniche algoritmiche molto efficienti su tipiche istanze del problema per il quale forniscono risposte.

- GareyJohnson-Cap1-parte04.mp4 commenti l'ultima parte della [GJ79, Sezione 1.3]. Insiste sulla rilevanza di usare codifiche *ragionevoli* delle strutture dati che vengono manipolate dagli algoritmi e di modelli di calcolo altrettanto ragionevoli.

7.2.3 Macchine di Turing Deterministiche DTM e classe PTime

Per parlare di complessità degli algoritmi e di riduzioni tra problemi (computazionali) non solo serve accordarsi sulla nozione (intuitiva) di *rappresentazione ragionevole dell'input*, ma è necessario fissare un modello di calcolo, cioè un interprete meccanico di riferimento in grado di eseguire gli algoritmi forniti per risolvere un qualsiasi problema, fissata una sua qualsiasi istanza.

- DTM-e-PTIME.mp4 ricorda la definizione del modello di computazione Macchine di Turing Deterministiche DTM e la classe di complessità computazionale associata PTime

  DTM-e-PTIME.pdf è il documento di riferimento.

- DTM-somma-numeri-binari.mp4 illustra la progettazione praticamente completa di una DTM che somma due numeri naturali in notazione binaria.

  DTM-somma-numeri-binari.pdf è il documento di riferimento.

7.2.4 Macchine di Turing Non deterministiche NDTM e classe NPTime

Oltre che di DTM e di *rappresentazione ragionevole dell'input*, per parlare di complessità degli algoritmi e di intrattabilità, è necessario fissare un ulteriore modello di calcolo, il cui scopo, intuitivamente, è comprimere il tempo necessario a esplorare alternative possibili che si presentano quando si cerca di risolvere un problema, data una sua istanza. Il modello computazionale che comprime il tempo sono le Macchine di Turing Non deterministiche.

- NDTM-e-NPTIME.mp4 ricorda la definizione del modello di computazione Macchine di Turing Non deterministiche NDTM e la classe di complessità computazionale associata NPTime

  NDTM-e-NPTIME.pdf è il documento di riferimento.

- NDTM-sottoinsiemi-somma-uguale.mp4 illustra la progettazione a blocchi di una NDTM che risolve il seguente problema *decisionale*:

  Definizione 7.2.1 Sia dato un insieme S di numeri naturali. Determinare l'esistenza di $T \subseteq S$ tale che $\sum_{x \in T} x = \sum_{y \in S/T} y$. Il risultato del problema è, quindi, una tra le due possibili risposte: "Sì, T esiste", oppure, "No, T non esiste". □

  NDTM-sottoinsiemi-somma-uguale.pdf è il documento di riferimento.

7.3 PTime e NPTime "senza Macchine di Turing"

Le Macchine di Turing, deterministiche e non, sono "troppo **concrete**". In generale siamo interessati a parlare di algoritmi su strutture dati, come grafi, insiemi, liste, ad esempio, che rappresentano l'informazione in maniera più trasparente rispetto ad insiemi di stringhe in $\{0,1\}^*$.

- PTime-NPTime-e-linguaggi-standard-paradigmatici.mp4 parla di classi PTime e NPTime in termini alternativi a quelli classici, cioè per mezzo di linguaggi imperativi *paradigmatici* dotati di strutture sintattiche minimali:

  LID Sta per *Linguaggio Imperativo Deterministico*. Esso dà la possibilità di assegnare a variabili valori calcolati con espressioni aritmetiche usuali, ma solo su numeri naturali, e fornisce costrutti tipici per la programmazione strutturata: sequenza, selezione, iterazione.

LIND Sta per *Linguaggio Imperativo Non Deterministico*. Estende LID con un costrutto oracolare che abbiamo chiamato `scelta(n)` in cui `n` è un numero intero.

La tipica configurazione in cui immaginare di usare `scelta(n)` è:

```
// x1 ... x ... xM  configurazione
x <-- scelta(n) // assegna ad x un valore in [0,n)
istruzione1
..
istruzioneN
```

L'interpretazione su cui ci accordiamo stabilisce che, a seguito della interpretazione della assegnazione `x <-- scelta(n)`, si creano `n` processi `P1`, ..., `Pn`; ciascuno di essi esegue la sequenza:

```
istruzione1
..
istruzioneN
```

ma `P1` usa `x` con valore `0`, `P2` usa `x` con valore `1` e così via fino all'ultimo processo che usa `x` con valore `n-1`.

Il messaggio principale è che in un linguaggio come LID, quindi come LIND, siamo liberi di assumere l'esistenza di costrutti sintattici ragionevoli per descrivere strutture dati come grafi, formule logiche, etc..

PTime-NPTime-e-linguaggi-standard-paradigmatici.pdf è il documento di riferimento nel quale si illustra una brevissima computazione di programma in LIND che risolve non deterministicamente una semplice istanza di KP, ma espresso in forma decisionale.

- Overhead-polinomiale.mp4 fornisce la giustificazione del perché è possibile usare LID, o LIND, al posto delle Macchine di Turing per definire le classi PTime e NPTime, pur permettendo l'uso esplicito ed ovvio di strutture dati come grafi, insiemi, tuple, etc.:

  – Da un lato possiamo sempre assumere che le strutture dati manipolate da un algoritmo scritto in un linguaggio di programmazione imperativo come LID, o LIND, sono sempre in stringhe del linguaggio $\{0,1\}^*$ con un costo non eccessivo, cioè polinomiale.

  Esempio 7.3.1 Ad esempio, un grafo (E,V) esprimibile in LID, o LIND, può essere sempre convertito in una stringa del linguaggio $\{0,1\}^*$, attraverso una funzione e, in modo che la dimensione $|(E,V)|$ del grafo sia tale che $|(E,V)| \leq p(|e(E,V)|)$, per un qualche polinomio $p(x)$. □

  – Dall'altro, possiamo sfruttare il fatto che ogni operazione in LID, o LIND, è simulabile da una Macchina di Turing in tempo polinomiale.

Globalmente, quindi, lavorare su rappresentazioni intuitive e ragionevoli di strutture dati attraverso algoritmi scritti in LID, o LIND, è come lavorare

indirettamente sulla loro rappresentazione binaria che avverrebbe attraverso macchine di Turing:

> “Si dice che i due modelli sono equivalenti a meno di *overhead* polinomiale, cioè a meno di un sovraccarico computazionale nel passare da un modello all’altro che è sempre limitato da un opportuno polinomio.”

Overhead-polinomiale.pdf è il documento di riferimento.

7.4 “Equivalenza” tra ottimizzazione e decisione

Finora abbiamo parlato di problemi di ottimizzazione, con particolare riferimento a KP, il cui scopo è scegliere opportuni insiemi di valori di variabili per massimizzare, o minimizzare, una qualche funzione obiettivo, sottostando ad opportuni vincoli.

La teoria della NP-*completeness* si è sviluppata in relazione alla complessità di problemi *decisionali*, non di ottimizzazione.

Molto brevemente, il motivo nasce da come si è evoluta la teoria degli automi, il cui scopo è definire un linguaggio come sottoinsieme di $\Sigma^* = \{0,1\}^*$ cui appartengono tutte le sequenze di lunghezza arbitraria composte di soli 0 e 1. Si identifica formalmente un linguaggio $\mathcal{L} \subseteq \Sigma^*$ attraverso l’automa in grado di *decidere* se $w \in \mathcal{L}$, per ogni $w \in \Sigma^*$. I linguaggi più semplici sono definiti per mezzo di automi che non hanno bisogno di alcuna memoria, oltre a quella rappresentata dagli stati che li compongono, per esempio. I linguaggi più complessi richiedono, invece, la potenza computazionale delle Macchine di Turing che sfruttano il nastro, cioè una memoria senza limitazioni sul numero di posizioni disponibili a memorizzare dati, per eseguire tutto quanto è necessario ad accettare, o meno, una stringa in input, rendendole il modello di calcolo che formalizza quanto di più difficile sappiamo fare dal punto di vista computazionale.

È quindi naturale valutare la complessità di un problema in termini della complessità dell’algoritmo (automa regolare, automa *push-down*, Macchina di Turing, ad esempio) che *guarda al problema come se fosse un linguaggio da decidere*: si risponde “Sì” se un dato input vi appartiene, si risponde “No” altrimenti. Quanto segue descrive come dimostrare che decidere è complesso quando ottimizzare.

- Se-decido-allora-ottimizzo.mp4 illustra:
  - come impostare formalmente la relazione tra due problemi in modo da stabilire quando uno non è più difficile da risolvere dell’altro;
  - come possiamo concludere che KP non è più difficile di un problema decisionale naturale che gli possiamo associare.

    In particolare, data l’istanza $((w_1,\dots,w_n),(p_1,\dots,p_n),C)$ di KP, possiamo indicare con $((w_1,\dots,w_n),(p_1,\dots,p_n),C,l)$ l’istanza del problema decisionale $\mathsf{KP}(l,C)$ corrispondente definito come:

$$\sum_{k=1}^{n} p_i \cdot x_i^* \geq l \sum_{k=1}^{n} w_i \cdot x_i^* \qquad\qquad \leq C \ .$$

Allora, se, per ogni l, supponiamo di saper rispondere a $\mathsf{KP}(l, C)$, risolviamo KP associato, cercando il *più piccolo valore* di l per cui esiste $\overline{x}^* = (x_1^*, \ldots, x_n^*)$ tale che:

$$\sum_{k=1}^{n} p_i \cdot x_i^* < l + 1$$
$$\sum_{k=1}^{n} p_i \cdot x_i^* \geq l \ ,$$

ovviamente a patto che $\sum_{k=1}^{n} w_i \cdot x_i^* \leq C$.

Se-decido-allora-ottimizzo.pdf è il documento di riferimento.

- Se-ottimizzo-allora-decido.mp4 completa il discorso, illustrando quanto segue:
  - sapendo risolvere un problema di ottimizzazione, non è più difficile risolvere un problema di decisione associato ad esso, semplicemente scambiando il ruolo dei due problemi discussi al punto precedente.
  - la semplicità dell'algoritmo decisionale per $\mathsf{KP}(l, C)$, dato un algoritmo per risolvere il problema di ottimizzazione KP.

  Se-ottimizzo-allora-decido.pdf è il documento di riferimento.

Presentazione alternativa alle Sezioni 7.3, 7.4. Per completezza, Theory-of-NP-Completeness.pdf è l'estratto di una prima parte de [GJ79, Capitolo 2]. Tratta le parole chiave "*Decisione Problems*", "*Languages*" e "*Econding Schemes*", con particolare enfasi sull'ultimo concetto, in un ordine leggermente diverso da quello scelto per le dispense.

7.5 Riduzione polinomiale

Riduzione-plinomiale.mp4 riassume il significato del concetto tecnico: "Ridurre un problema computazionale P_1 ad un problema computazionale P_2", che si può descrivere intuitivamente per mezzo di affermazioni intuitive come le seguenti, tra loro essenzialmente equivalenti:

- se è noto come risolvere P_2, allora è possibile risolvere P_1 grazie ad una trasformazione da P_1 a P_2 dal basso costo computazionale;
- tra le istanze di P_2 possiamo trovare tutte le istanze di P_1, a meno di una trasformazione da P_1 a P_2 dal basso costo computazionale;
- se non è noto come risolvere efficientemente P_1, allora non può essere noto come risolvere P_2 perché esistono istanze di P_1 in P_2 a meno della trasformazione che realizza la riduzione.

Queste affermazioni derivano dalla definizione formale di "Riduzione tra problemi decisionali P_1 e P_2", denotata come "$\leq_{\mathsf{P}}$".

Avendo noi l'obiettivo di scrivere nel dettaglio la sequenza di riduzioni

$$\begin{aligned} &\mathsf{SAT} \leq_{\mathsf{P}} \mathsf{NNF} \leq_{\mathsf{P}} \mathsf{CNF} \leq_{\mathsf{P}} \mathsf{3CNF} \\ &\leq_{\mathsf{P}} \mathsf{3COL} \leq_{\mathsf{P}} \mathsf{EXCO} \leq_{\mathsf{P}} \mathsf{Subsetsum} \leq_{\mathsf{P}} \mathsf{KP} \ , \end{aligned} \tag{7.1}$$

illustriamo la "Riduzione tra problemi decisionali P_1 e P_2" come sequenza di tre fasi.

Definizione 7.5.1 (Riduzione tra problemi computazionali P_1 e P_2) Il problema computazionale P_1 si riduce al problema computazionale P_2, e scriviamo $P_1 \leq_{\mathsf{P}} P_2$, se possiamo definire una opportuna *funzione di riduzione* $F : \mathrm{Dom}(P_1) \to \mathrm{Dom}(P_2)$ in cui $\mathrm{Dom}(P_1)$ e $\mathrm{Dom}(P_2)$ sono i domini che include come sottoinsieme le istanze di P_1 e P_2.

In particolare, $\mathrm{Dom}(P_1)$ e $\mathrm{Dom}(P_2)$ possono essere insiemi di grafi, tuple numeriche esplicite, etc., ed F è tale che:

1. *ha* costo polinomiale nella dimensione dell'input preso in $\mathrm{Dom}(P_1)$. Questo significa che, per ogni $x \in \mathrm{Dom}(P_1)$, il costo dell'algoritmo (in LID, o nella Macchina di Turing Deterministica corrispondente) è limitato superiormente dal valore $p(|x|)$, per un qualche polinomio fissato $p(x)$ che dipende solo dal problema P_1 da ridurre;

2. è possibile dimostrare che, se $x \in \mathrm{Dom}(P_1)$ e x *è un'istanza* di P_1 (se A_{P_1} è l'algoritmo che risolve P_1, allora "$A_{P_1}(x) =$ Sì"), allora $F(x)$ *è un'istanza* di P_2 (se A_{P_2} è l'algoritmo che risolve P_2, allora "$A_{P_2}(F(x)) =$ Sì");

3. è possibile dimostrare che, se $x \in \mathrm{Dom}(P_1)$ e x *non è un'istanza* di P_1 (se A_{P_1} è l'algoritmo che risolve P_1, allora "$A_{P_1}(x) =$ No"), allora $F(x)$ *non è un'istanza* di P_2 (se A_{P_2} è l'algoritmo che risolve P_2, allora "$A_{P_2}(F(x)) =$ No"). □

Proprietà 7.5.1 (Proprietà delle riduzioni) Dati i problemi P_1, P_2 e P_3:

1. se $P_1 \leq_{\mathsf{P}} P_2$ e $P_2 \leq_{\mathsf{P}} P_3$, allora $P_1 \leq_{\mathsf{P}} P_3$, cioè la relazione $\leq_{\mathsf{P}}$ è transitiva;

2. se per P_1 esiste un algoritmo in PTime e $P_2 \leq_{\mathsf{P}} P_1$, allora anche per P_2 esiste un algoritmo in PTime.

   Equivalentemente, ed in termini più "qualitativi", questo significa che se $P_2 \leq_{\mathsf{P}} P_1$ e per P_1 esiste un buon algoritmo, allora esiste un buon algoritmo anche per P_2. □

Il Punto 2 della **Proprietà 7.5.1** implica:

Corollario 7.5.1 Se $P_1 \leq_{\mathsf{P}} P_2$ e P_1 è intrattabile, cioè non è noto alcun algoritmo in PTime per esso, allora anche P_2 è intrattabile. □

Riduzione-plinomiale.pdf è il documento di riferimento.

7.5.1 Perché la riduzione è per noi rilevante?

Molto in breve, ricordiamo che SAT è il problema che richiede di rendere vera una formula della Logica Proposizionale il cui valore di verità dipende da un insieme di variabili.

Non esiste alcun dubbio sul fatto che saper risolvere efficientemente SAT sarebbe di estrema utilità per via di tutte le applicazioni in cui SAT trova impiego.

L'impegno profuso nel risolvere SAT in maniera efficiente è pressoché non quantificabile, ma nessuno è riuscito nell'intento sinora.

Il nostro obiettivo è dimostrare che KP è un problema intrinsecamente difficile, o intrattabile.

La "Riduzione tra problemi" permette di costruire istanze di KP che, a meno di traduzioni, sono, in realtà, istanze di SAT, permettendo di affermare:

$$\mathsf{SAT} \textit{ è un sotto-problema di } \mathsf{KP}$$

cioè, in taluni casi, tentare di costruire una risposta ad un'istanza di KP è come tentare di produrre una risposta per un'istanza di SAT, che, come detto, non è necessariamente ovvio.

Tecnicamente, per dimostrare che KP è intrattabile, cioè che SAT si riduce a KP, può consistere nel mostrare l'esistenza della *catena di riduzioni* in (7.1).

KP-intrattabile.mp4 illustra la dimostrazione che KP (in forma decisionale) è intrattabile: ipotizzando sia che $\mathsf{PTime} \neq \mathsf{NPTime}$, sia che il problema computazionale SAT stia in NPTime[1], cioè SAT è intrattabile, allora anche KP è intrattabile perché esiste la catena di riduzioni (7.1) e la relazione di riducibilità è transitiva.

KP-intrattabile.pdf è il documento di riferimento.

[1] Sappiamo *verificare* in tempo polinomiale se una formula è soddisfacibile, ma non sappiamo *rendere* soddisfacibile una formula in tempo polinomiale.

Parte III

Complessità computazionale

Problema della soddisfacibilità (SAT) 8

8.1 Calcolo proposizionale e problema SAT

Calcolo-proposizionale-e-SAT.mp4 è un ripasso su calcolo proposizionale con l'obiettivo finale di introdurre il problema SAT:

Formule proposizionali. Esse appartengono al linguaggio pf che include variabili logiche $x, x_1, \ldots, y, y_1, \ldots$, le costanti `true` e `false`, e tutte le formule costruite da $f_1, f_2 \in$ pf, usando gli operatori logici congiunzione, disgiunzione e negazione.

Insieme delle variabili di una formula. Per convenzione, indichiamo con $\mathrm{FV}(f)$ l'insieme di variabili su cui è costruita $f \in$ pf. L'insieme è definito induttivamente sulla struttura di f;

Assegnazione di valori di verità ad una formula. Fissata una formula f, possiamo fissare una funzione ϕ che ad ogni variabile in $\mathrm{FV}(f)$ assegna un valore in $\{\texttt{true}, \texttt{false}\}$. Se $\mathrm{FV}(f)$ ha n elementi, esistono 2^n funzioni ϕ distinte. Con Φ_ϕ indichiamo la funzione definita sulla struttura di una qualsiasi formula, che ne valuta il valore di verità una volta fissati i valori di verità delle variabili in $\mathrm{FV}(f)$ tramite ϕ.

Definizione 8.1.1 (Problema SAT) Data $f \in$ pf, con variabili $\mathrm{FV}(f) = \{x_1, \ldots, x_n\}$, è vero che esiste un'assegnazione di valori di verità ϕ alle variabili in $\mathrm{FV}(f)$ che *soddisfa* f, cioè tale che $\Phi_\phi(f) = \texttt{true}$? □

Calcolo-proposizionale-e-SAT.pdf è il documento di riferimento.

Note 8.1.1 Garey&Johnson-SAT.mp4 commenta brevemente la [GJ79, Sezione 3.1.1]. Il testo *non* definisce SAT sull'intero linguaggio pf, sul sottoinsieme di pf costituito da formule in *forma normale congiuntiva* la cui struttura definiremo in seguito.

Garey&Johnson-SAT.pdf è il documento di riferimento. □

8.2 SAT sta in NPTime ∩ NP-complete

SAT-sta-in-NP.mp4 completa il richiamo su SAT.

- Tramite un semplice algoritmo in LIND testimonia in modo intuitivo l'appartenenza di SAT alla classe NPTime. Dalla simulazione del semplice esempio dovrebbe essere evidente che ogni ramo termina ed è lungo tanto quante sono le variabili nella formula in input. In più, il costo della valutazione di f è proporzionale al numero di foglie, cioè di variabili, i cui nomi possono essere rappresentati da stringhe binarie.
- Ricorda anche che SAT è tra i problemi "più difficili" di NPTime, cioè tra quelli che incorporano la difficoltà intrinseca di risolvere efficientemente i problemi in NPTime. Il motivo tecnico di questa affermazione è il seguente:

  > "Ogni problema in NPTime è riducibile a SAT [Cook–Levin theorem]."

  La proprietà è notevole per via di due conseguenze:

  - ogni problema P cui SAT è riducibile è automaticamente NP-complete, cioè A e SAT inglobano entrambi la difficoltà intrinseca che li rende intrattabili, ma sotto "spoglie" formali diverse;
  - se esiste un algoritmo efficiente per risolvere un qualsiasi problema NP-complete, allora esiste per tutti i problemi in NPTime, da cui conseguirebbe PTime = NPTime.

SAT-sta-in-NP.pdf è il documento di riferimento.

Riduzione SAT $\leq_P$ NNF 9

9.1 Problema NNF

NNF-problem.mp4 introduce il problema *Negation Normal Form* (NNF), definendo il linguaggio delle formule in *forma normale negata*.

***Formule* nnf.** Appartengono al linguaggio che include *letterali* $x, \overline{x}, x_1, \overline{x}_1, \ldots, y$, $\overline{y}, y_1, \overline{y}_1, \ldots$, le costanti `true` e `false`, e tutte le formule costruite da $f_1, f_2 \in$ nnf usando gli operatori logici congiunzione, disgiunzione, ma *non la negazione*.

***Insieme delle variabili in formule di* nnf.** Come per le formule pf.

***Assegnazione di valori di verità a formule di* nnf.** Come per le formule pf.

Definizione 9.1.1 (Problema NNF) Data $f \in$ nnf, con variabili $\mathrm{FV}(f) = \{x_1, \ldots, x_n\}$, è vero che esiste un'assegnazione di valori di verità ϕ alle variabili in $\mathrm{FV}(f)$ che *soddisfa* f, cioè tale che $\Phi_\phi(f) =$ `true`? □

NNF-problem.pdf è il documento di riferimento.

9.2 Funzione di riduzione M : pf $\to$ nnf

Vogliamo definire M : pf $\to$ nnf per la quale riusciamo a dimostrare che $f \in$ SAT se, e solo se, $M(f) \in$ NNF.

SATtoNNF-funzione-di-riduzione.mp4 illustra come le leggi di De Morgan possono essere usate per definire la *funzione di riduzione* M : pf $\to$ nnf, cioè un algoritmo che trasforma una qualsiasi $f \in$ pf in $M(f) \in$ nnf. L'intuizione è che $M(f)$ "spinge" le negazioni eventualmente presenti in f verso i nomi delle variabili di f.

SATtoNNF-funzione-di-riduzione.pdf è il documento di riferimento.

9.3 Dimostrazione

SATtoNNF-prologo-alla-dimostrazione.mp4 illustra a che punto siamo nel processo di costruire la nostra prima riduzione.

Il video dal titolo SATtoNNF-prologo-alla-dimostrazione.mp4, nel quale avrei voluto registrare la dimostrazione che ha SATtoNNF-dimostrazione.pdf come riferimento, è saltata per un errore nell'uso dell'interfaccia.

- SATtoNNF-Lemma1.mp4 dimostra il primo dei due lemmi, cioè che la funzione di riduzione M preserva le variabili di ogni formula cui è applicata.

  SATtoNNF-Lemma1.pdf è il documento di riferimento.

- SATtoNNF-Lemma2.mp4 dimostra il secondo dei due lemmi, cioè che la funzione di riduzione M preserva la funzione di interpretazione, cioè che $\Phi_\phi(M(f)) = M(\Phi_\phi(f))$.

  SATtoNNF-Lemma2.pdf è il documento di riferimento.

10.1 Formule in *conjunctive normal form* e CNF

Conjunctive-normal-forms-e-CNF.mp4, illustrando in che contesto ci poniamo l'obiettivo di dimostrare NNF $\leq_{\mathsf{P}}$ CNF, rammenta la struttura delle formule in cnf.

Definizione 10.1.1 (*Conjunctive normal form*) Il linguaggio (o insieme) cnf delle *conjunctive normal form* è definito dalla seguente grammatica:

$$\begin{aligned} F &::= [C] \mid [C], F && \text{(formule cnf)} \\ C &::= \texttt{true} \mid \texttt{false} \mid x \mid \overline{x} \mid C, C && \text{(calusole)} \ , \end{aligned} \tag{10.1}$$

in cui C è lo *start symbol* della grammatica che definisce le clausole.

La grammatica segue la convenzione notazionale secondo cui il simbolo "," tra clausole corrisponde alla congiunzione logica "$\wedge$", mentre essa corrisponde alla disgiunzione logica "$\vee$" all'interno delle clausole. □

Conjunctive-normal-forms-e-CNF.pdf è il documento di riferimento.

10.2 Trasformazione di Tseitin (Funzione di riduzione)

La dimostrazione di NNF $\leq_{\mathsf{P}}$ CNF è basata sulla trasformazione T : nnf $\to$ cnf di Tseytin. Essa assicura che $f \in$ nnf è soddisfacibile esattamente quando $T(f) \in$ cnf lo è.

Una possibile guida all'intuizione che giustifica la definizione di T è vedere ogni $f \in$ nnf come composizione di formule, ciascuna isolata dalle altre, ma in grado di comunicare attraverso canali rappresentati da equivalenze logiche; introdurre equivalenze logiche in f attraverso T è la chiave per la trasformazione in clausole a basso costo computazionale che preserva la soddisfacibilità.

NNFtoCNF-trasformazione-T-intuizione.mp4 dà conto dei principi intuitivi che guidano la definizione della trasformazione di Tseytin. Il punto cruciale di $T(f)$ è vedere f come circuito in cui ogni connessione "fisica" tra le porte ha un nome che assume il

significato di variabile logica e può essere messo in equivalenza logica con disgiunzioni, congiunzioni o nomi di altre variabili. *Le equivalenze logiche corrispondono alle clausole da costruire.*
NNFtoCNF-trasformazione-T-intuizione.pdf è il documento di riferimento.

NNFtoCNF-trasformazione-T-definizione.mp4 definisce ricorsivamente $T : \mathsf{nnf} \to \mathsf{cnf}$ sulla struttura della formula f cui è applicata; T si appoggia ulteriormente ad una funzione $C : (\text{Variabili} \times \mathsf{nnf}) \to \mathsf{cnf}$ cui è demandato effettivamente il compito di tradurre formule in nnf. Intuitivamente, T ha complessità lineare nella dimensione di f perché aggiunge un nome nuovo di variabile ed un operatore logico "$\Leftrightarrow$" per ogni sotto-formula di f. Al termine $|T(f)| \in O(|f|)$.
NNFtoCNF-trasformazione-T-definizione.pdf è il documento di riferimento.

NNFtoCNF-trasformazione-T-correttezza.mp4 illustra la dimostrazione della correttezza di $T : \mathsf{nnf} \to \mathsf{cnf}$, ovvero che la forma di $T(f)$ è un elemento di cnf in accordo con la grammatica (10.1).
NNFtoCNF-trasformazione-T-correttezza.pdf è il documento di riferimento..

10.2.1 Possibile materiale integrativo

- Tseytin transformation (Wikipedia) rende disponibile On the Complexity of Derivation in Propositional Calculus, l'articolo scientifico originale di Tseytin;
- Risorsa on-line "Tseytin transform" by Prof. Hans Zantema (Coursera).

10.3 Dimostrazione di NNF $\leq_P$ CNF

NNFtoCNF-Enunciato-e-dimostrazione.mp4 enuncia il significato di dimostrare NNF $\leq_P$ CNF ed illustra che la dimostrazione è basata sulla trasformazione di Tseytin ed assume l'aver dimostrato quel che chiamiamo LemmaT.
NNFtoCNF-Enunciato-e-dimostrazione.pdf è il documento di riferimento.

10.3.1 Dimostrazione del Lemma T

Proposition 10.3.1 (LemmaT) Sia $f \in \mathsf{nnf}$. Per brevità notazionale, siano $V_f = \mathrm{FV}(f)$ (insieme delle variabili di f) e $T_f = \{a, a_1, \ldots, a_n\}$ l'insieme di variabili di $T(f)$ *nuove* rispetto a V_f, cioè tali che $\{a, a_1, \ldots, a_n\} \cap V_f = \emptyset$. Inoltre, assumiamo che a è tale che $T(f) = [a], C[a, f]$, cioè la variabile nuova rispetto a V_f che corrisponde alla radice della formula $T(f)$. Allora, per ogni funzione di valutazione ϕ definita su V_f:

$$\exists \phi, \Phi_\phi(f) = \texttt{true} \Leftrightarrow \exists \phi'.(\Phi_{\phi'}(C[a, f]) = \texttt{true} \wedge \Phi_{\phi'}(a) = \texttt{true}) \tag{10.2}$$

$$\exists \phi, \Phi_\phi(f) = \texttt{false} \Leftrightarrow \exists \phi'.(\Phi_{\phi'}(C[a, f]) = \texttt{true} \wedge \Phi_{\phi'}(a) = \texttt{false}) \tag{10.3}$$

in cui ϕ' è una funzione di valutazione definita sulle variabili in T_f tale che $\phi'(x) = \phi(x)$ per ogni $x \in V_f$, cioè, ϕ' coincide con ϕ sulle variabili di f. □

La dimostrazione del LemmaT procede per induzione strutturale su f, simultaneamente su entrambi i predicati (10.2) e (10.3).

NNFtoCNF-LemmaT-dimostrazione-Punto1.mp4 sviluppa passo passo la dimostrazione del Punto (10.2) nel LemmaT (predicato 10.3.1).
NNFtoCNF-LemmaT-dimostrazione-Punto1.pdf è il documento associato.

Note 10.3.1 NNFtoCNF-Induzione-simultanea-Schema.mp4 illustra lo schema mentale da adottare per seguire in dettaglio la dimostrazione per induzione simultanea del Lemma T. In particolare, dati due predicati generici:

$$A(f) \Rightarrow B(f) \tag{10.4}$$
$$C(f) \Rightarrow D(f) \tag{10.5}$$

in cui $A(f), B(f), C(f)$ e $D(f)$ sono proprietà (predicati) di f, descrive come la dimostrazione di (10.4) può basarsi su ipotesi induttive che implicano, simultaneamente, la verità di (10.4) e di (10.5), ma su componenti di f. La corrispondenza tra (10.2) e (10.4), e tra (10.3) e (10.5) dovrebbe risultare ragionevolmente evidente.
NNFtoCNF-Induzione-simultanea-Schema.pdf è il documento di riferimento. □

NNFtoCNF-LemmaT-dimostrazione-Punto2.mp4 non è stato sviluppato per mancanza di tempo, ma soprattutto di energie. Avrebbe sviluppato passo passo la dimostrazione del Punto (10.3) nel LemmaT (predicato 10.3.1).
NNFtoCNF-LemmaT-dimostrazione-Punto2.pdf è il documento associato.

11.1 Problema 3CNF

CNFto3CNF-definizione-3cnf-e-3CNF.mp4 richiama sia la grammatica per il linguaggio 3cnf delle formule proposizionali in *conjunctive normal form con esattamente 3 letterali*, sia il problema associato 3CNF.

CNFto3CNF-definizione-3cnf-e-3CNF.pdf è il documento di riferimento.

11.2 Funzione di riduzione

Per seguire senza confusioni video e documenti qui sotto disponibili occorre tenere presente quanto già segnalato in **Note 8.1.1**, che si riferisce a Garey&Johnson-SAT.pdf ([GJ79, Sezione 3.1.1]), e che riportiamo qui per convenienza:

- Il testo di riferimento parla di una riduzione per dimostrare SAT $\leq_P$ 3CNF. Se così fosse, il dominio della funzione di riduzione dovrebbe essere costituito dall'insieme pf delle formule proposizionali.
- In realtà il testo di riferimento descrive una riduzione per dimostrare CNF $\leq_P$ 3CNF il cui dominio è costituito dall'insieme cnf delle formule proposizionali in *conjunctive normal form*.

CNFto3CNF-riduzione-definizione.mp4, in analogia con Garey&Johnson-SAT.pdf ([GJ79, Sezione 3.1.1]), ma non in maniera identica, introduce la riduzione $F : \mathsf{cnf} \to \mathsf{3cnf}$ da formule in *conjunctive normal form* a formule in *conjunctive normal form con esattamente 3 letterali*.

CNFto3CNF-riduzione-definizione.pdf è il documento di riferimento.

CNFto3CNF-riduzione-esempio.mp4 illustra un modo per derivare la non soddisfacibilità di $F([x], [\overline{x}]) \in \mathsf{3cnf}$ dalla non soddisfacibilità di $[x], [\overline{x}] \in \mathsf{cnf}$.

CNFto3CNF-riduzione-esempio.pdf è il documento di riferimento.

11.3 Dimostrazione

CNFto3CNF-dimostrazione.mp4 illustra la dimostrazione di $\mathsf{CNF} \leq_{\mathsf{P}} \mathsf{3CNF}$ che equivale a dimostrare la seguente:

Proprietà 11.3.1 ($\mathsf{CNF} \leq_{\mathsf{P}} \mathsf{3CNF}$) Per ogni $f \in \mathsf{cnf}$ esiste un'assegnazione ϕ di valori di verità alle variabili in $\mathrm{FV}(f)$ di f, tale che $\Phi_\phi(f) = \texttt{true}$ se, e solo se, esiste una assegnazione ϕ' di valori di verità alle variabili in $\mathrm{FV}(F(f))$ che estende ϕ e tale che $\Phi_\phi(F(f)) = \texttt{true}$. □

La dimostrazione procede per induzione sia sul numero n di clausole in f, quando $n > 1$, sia sul numero di letterali dell'unica clausola di f, quando $n = 1$.

CNFto3CNF-dimostrazione.pdf è il documento di riferimento.

Riduzione 3CNF $\leq_P$ 3COL 12

12.1 Problema 3COL

Un ulteriore passo per dimostrare SAT $\leq_P$ KP è la riduzione 3CNF $\leq_P$ 3COL. La riduzione fa corrispondere la soddisfacibilità di formule di 3cnf alla possibilità di costruire grafi che stanno nell'insieme 3col di grafi *3-colorabili*.

Quindi, il dominio della funzione della riduzione è 3cnf, mentre il codominio è graph, i cui elementi sono coppie (V, E), con V l'insieme dei vertici ed E insieme degli archi. Un sottoinsieme di graph è ovviamente 3col.

3COL-definizione.mp4 richiama il problema 3COL.

Definizione 12.1.1 Sia $G = (V, E)$ in graph. Esiste una funzione $C : V \to \{v, r, b\}$, tale che $C(V)$ colora i vertici in V in modo che, per ogni coppia (x, y) di vertici adiacenti, $C(x) \neq C(y)$? □

3COL-definizione.pdf è il documento associato.

12.2 Funzione di riduzione

3CNFto3COL-TrasformazioneL.mp4 definisce la funzione di riduzione L : 3cnf $\to$ graph:

- in prima battuta, L costruisce un grafo $L(\mathcal{L}(f))$ che rappresenta l'insieme $\mathcal{L}(f)$ dei letterali nella formula $f \in$ 3cnf.

  Inizialmente, $L(\mathcal{L}(f))$ permette una corrispondenza molto lasca tra assegnazione di valori di verità che soddisfano la formula f e la 3-colorabilità di $L(\mathcal{L}(f))$. Ad esempio, è ammesso assegnare identico colore ai nodi in $L(\mathcal{L}(f))$ che corrispondono ad un letterale l ed al suo complementare $\bar{l}$; la libertà di assegnazione di colori è facilmente limitabile, aggiungendo archi tra il nodo corrispondente ad l ed il nodo del complementare $\bar{l}$ in $L(\mathcal{L}(f))$.

  Tale limitazione non è tuttavia sufficiente per una piena corrispondenza tra soddisfacibilità di una formula e 3-colorabilità;

- in seconda battuta,la funzione L è estesa per tradurre clausole in *gadget* la cui struttura è "calibrata" per limitare la libertà con cui colorare i nodi del grafo ottenuto; ovviamente, lo scopo è far corrispondere la soddisfacibilità di una clausola alla 3-colorabilità di un *gadget*.

- illustra un esempio non del tutto banale di più colorazioni per uno stesso grafo prodotto a partire da una formula in 3cnf.

3CNFto3COL-TrasformazioneL.pdf è il documento associato.

12.3 Dimostrazione

3CNFto3COL-Proprietà1.mp4 discute le 7 colorazioni possibili di un *gadget* prodotto da $L(f)$, assumendo che almeno uno dei nodi che codifica un letterale sia verde, come conseguenza del fatto che il suo valore di verità è `true`.
3CNFto3COL-Proprietà1.pdf è il documento di riferimento.

3CNFto3COL-Proprietà2.mp4 illustra l'impossibilità di colorare $F(f)$ se f è insoddisfacibile, cioè se non esiste alcuna assegnazione di valori di verità per f che la rende vera. In questa situazione esiste sempre una clausola C in cui tutti i letterali sono `false`. Ad essi corrispondono tre nodi dello stesso colore che propagano l'impossibilità di colorare la parte di grafo che codifica C.
3CNFto3COL-Proprietà2.pdf è il documento di riferimento.

3CNFto3COL-Dimostrazione.mp4 enuncia il predicato $f \in \mathsf{3cnf} \Leftrightarrow L(f) \in \mathsf{3col}$, con cui dimostrare $\mathsf{3CNF} \leq_{\mathsf{P}} \mathsf{3COL}$, e illustra la strategia di dimostrazione, assumendo d'aver disponibili due lemmi che, per convenzione, chiamiamo le Proprietà1 e Proprietà2.
3CNFto3COL-Dimostrazione.pdf è il documento di riferimento.

12.4 Materiale integrativo (eventuale)

SAT to 3COL, by Albert Meyer.mp4 ex professore al MIT, illustra una riduzione da SAT a 3COL, definita in collaborazione il suo studente di dottorato Larry Stockmeyer.

Reducibility among combinatorial problems.pdf è una riedizione del 2010 dell'articolo originale di [Kar72]. L'introduzione della riedizione è una panoramica del processo che ha portato Karp:

- sia a riassumere la rilevanza del concetto di riduzione, evitando di doversi rifare al problema decisionale di riconoscimento di un linguaggio da parte di una macchina di Turing;

- sia alle catene di riduzione che scrive.

Per ridurre SAT a 3COL, Karp segue una strada che non è né quella che stiamo seguendo in queste seconda parte del corso, né quella diretta suggerita da Meyer, ma è quella che si trova in [GJ79, Capitolo 3].

Karp-Reducibility-Albero.pdf è il documento con la sola pagina del lavoro di Karp, contenente l'albero di riduzioni.

12.4.1 Esercizio

Siano date le formule $A = \overline{x \wedge \overline{x}}$ e $B = \overline{\overline{x} \vee x}$.

Verificare che la soddisfacibilità di A e B è mantenuta in $L(F(T(M(A))))$ e $L(F(T(M(B))))$, in cui $M : \mathsf{pf} \to \mathsf{nnf}$, $T : \mathsf{nnf} \to \mathsf{cnf}$, $F : \mathsf{cnf} \to \mathsf{3cnf}$ e $L : \mathsf{3cnf} \to \mathsf{3col}$.

13.1 Problema EXCO

3COLtoEXCO-Introduzione.mp4 definisce il problema EXCO che, per comodità, richiamiamo qui sotto, e ne illustra alcuni esempi e contro-esempi.

Definizione 13.1.1 (Problema *Exact Cover* EXCO) Siano dati U, insieme finito che chiamiamo *universo*, e $S \subseteq \mathcal{P}(U)$ sottoinsieme dell'insieme delle parti di U. La coppia (U, S) è istanza del problema EXCO se e solo se esiste un sottoinsieme $S' \subseteq S$ che è una *copertura esatta* di U, ovvero:

$$U = \bigcup_{x \in S'} x \tag{13.1}$$

$$\emptyset = x \cap y \qquad \forall x, y \in S'.\ x \neq y\ . \tag{13.2}$$

□

3COLtoEXCO-Introduzione.pdf è il documento di riferimento.

13.1.1 Prologo alla funzione di riduzione

3COLtoEXCO-Intuizione-parte1.mp4 parte da un semplice grafo:

$$\begin{aligned} G &= (V, E) \\ V &= \{x, y, z\} \\ E &= \{(x, y), (y, z)\} \end{aligned} \tag{13.3}$$

in 3COL e costruisce passo passo una prima idea di come ottenere la coppia:

$$\begin{aligned} U = \{&v_x, v_{xy}, \ldots, v_y, v_{yx}, \ldots, v_z, v_{zy}, \ldots \\ &, b_x, b_{xy}, \ldots, b_y, b_{yx}, \ldots, b_z, b_{zy}, \ldots \\ &, r_x, r_{xy}, \ldots, r_y, r_{yx}, \ldots, r_z, r_{zy}, \ldots\} \end{aligned} \tag{13.4}$$

$$\begin{aligned} S = &\{\{v_x\}, \{b_x\}, \{r_x\}, \{v_y\}, \{b_y\}, \{r_y\}, \{v_z\}, \{b_z\}, \{r_z\}\} \\ &\cup \{\{b_x, b_{xy}\}, \{r_x, r_{xy}\}, \{v_x, v_{yx}\}, \{v_x, b_{yx}\}, \{v_x, r_{yx}\}, \ldots\} \subseteq \mathcal{P}(U) \end{aligned} \tag{13.5}$$

da G. L'universo U contiene nomi di punti sulla struttura a grafo di G tra i quali devono esistere opportune relazioni espresse come elementi di S; le relazioni fissano colorazioni compatibili tra nodi adiacenti.

Note 13.1.1 Non deve trarre in inganno che, per guidare l'intuizione, l'insieme S sia rappresentato come grafo in cui seguire cammini per individuare una colorazione del grafo di partenza. L'obiettivo è, ovviamente, il punto di arrivo nel quale possiamo immaginare di ignorare gli archi tra le componenti degli insiemi, ed apprezzare che è stato costruito un insieme di insiemi di punti di incontro tra colori compatibili. Quell'insieme di insiemi rappresenta S' ed è un sottoinsieme di un insieme più ampio che può contenere altri S' in grado di suggerire ulteriori colorazioni corrette, alternative a quella data. □

3COLtoEXCO-Intuizione-parte1.pdf è il documento di riferimento.

Esempio 13.1.1 (Da 3COL a EXCO) 3COLtoEXCO-Intuizione-parte2.mp4 costruisce tre possibili esempi di sottoinsieme S' di S, ciascuno individuato in accordo con una possibile colorazione del grafo di partenza; ogni S' soddisfa i due vincoli (13.1) e (13.2), seguendo la colorazione del grafo di partenza.

3COLtoEXCO-Intuizione-parte2.pdf è il documento di riferimento. □

13.1.2 L'intuizione ha un problema

3COLtoEXCO-Problema.mp4 illustra il motivo per cui l'intuizione sinora seguita per definire una trasformazione che, per ora, possiamo chiamare E, la quale ha graph come dominio e coppie $(U, S \subseteq \mathcal{P}) = E(G)$ come co-dominio, pur essendo sulla strada giusta, purtroppo rende vera l'implicazione:

$$\exists U, S, G.(U, S) \in \mathsf{EXCO} \wedge E(G) = (U, S) \not\Rightarrow G \in \mathsf{3COL} \ , \tag{13.6}$$

cioè, pur essendo (U, S) l'immagine di un qualche grafo G, ottenuta da un grafo tramite E, la copertura non suggerisce alcuna colorazione per G.

3COLtoEXCO-Problema.pdf è il documento di riferimento.

13.1.3 L'intuizione definitiva

Aggiorniamo (U, S) in (13.4) alla coppia (U^*, S^*) seguente:

$$U^* = \{x, y, z\} \cup U \tag{13.7}$$

$$\begin{aligned} S^* = (S \setminus \{&\{v_x\}, \{b_x\}, \{r_x\}, \{v_y\}, \{b_y\}, \{r_y\}, \{v_z\}, \{b_z\}, \{r_z\}\}) \\ \cup \{&\{x, v_x\}, \{x, b_x\}, \{x, r_x\} \\ &, \{y, v_y\}, \{y, b_y\}, \{y, r_y\} \\ &, \{z, v_z\}, \{z, b_z\}, \{z, r_z\}\} \end{aligned} \tag{13.8}$$

in cui U è esteso con tre nuovi elementi x, y e z, uno per ogni vertice di V, ed S^* include solo relazioni che potremmo definire "*non banali*", cioè almeno tra due elementi di U^*. Ad esempio, la relazione $\{x, v_x\}$ indica la specifica scelta del colore verde per il vertice x che non è più espressa dal semplice singoletto $\{v_x\}$ presente in S, ma assente da S^*.

- 3COLtoEXCO-adiacenti-colori-uguali.mp4, da un lato, illustra che, usando (U^*, S^*) in (13.8), l'assegnazione del verde a due vertici adiacenti x e y, per esempio, non permette una completa copertura dell'universo U.

  3COLtoEXCO-adiacenti-colori-uguali.pdf è il documento di riferimento.

- 3COLtoEXCO-singolo-colore.mp4, dall'altro, illustra che, usando (U^*, S^*) in (13.8), impedisce di concludere la colorazione di G se, ad esempio, $\{\{x, v_x\}, \{x, b_x\}\} \subset S' \subset S^*$, cioè se assegniamo due colori al vertice x.

  3COLtoEXCO-singolo-colore.pdf è il documento di riferimento.

13.2 Funzione di riduzione

3COLtoEXCO-Definizione-trasformazione.mp4 introduce la definizione della funzione di riduzione $E_x : \mathsf{graph} \to (X \times \mathcal{P}(X))$, per un qualche insieme X; E_x è tale che, se $E_x((V, E)) = (U, S)$, allora U è l'universo di elementi di cui considerare sottoinsiemi, ed S un sottoinsieme dell'insieme delle parti di U.

In particolare, E_x introduce elementi dell'universo U per identificare:

- i vertici in V;
- i tre colori che ciascun vertice può assumere;
- i punti di contatto dei colori, che, immaginandoli liquidi, si espandono dal vertice che colorano a tutti i vertici adiacenti al vertice colorato.
- gli elementi (che sono insiemi) in S che:
  - presi singolarmente, indicano colorazioni legali di nodi adiacenti
  - inclusi in un opportuno sottoinsieme, possono costituire una delle possibili coperture esaatte di U.

3COLtoEXCO-Definizione-trasformazione.pdf è il documento di riferimento.

13.3 "Dimostrazione"

La dimostrazione che $\mathsf{3COL} \leq_{\mathsf{P}} \mathsf{EXCO}$ è un'argomentazione più articolata di quella molto essenziale fornita da [Koz] e disponibile in 3COLtoEXCO-Kozen.pdf. In particolare:

- 3COLtoEXCO-Dimostrazione-Parte1.mp4 illustra il dettaglio della dimostrazione per l'implicazione $G \in \mathsf{3COL} \Rightarrow E_x(G) \in \mathsf{EXCO}$, in cui, come è intuibile, il ruolo giocato dall'ipotesi che a nodi adiacenti sono assegnati colori diversi da parte di una funzione $\chi : V \to \{v, b, r\}$ è fondamentale per stabilire le proprietà della copertura esatta S'.

  3COLtoEXCO-Dimostrazione-Parte1.pdf è il documento di riferimento;

- 3COLtoEXCO-Dimostrazione-Parte2.mp4 **non è stato sviluppato**, e non lo sarà. Avrebbe illustrato il dettaglio della dimostrazione per l'implicazione $G \in \mathsf{3COL} \Leftarrow E_x(G) \in \mathsf{EXCO}$. Essa procede per contraddizione, immaginando che da una copertura S' dell'universo sia possibile assegnare lo stesso colore a due vertici x ed y in V adiacenti, cioè tali che $(x, y) \in E$. Se così fosse, l'insieme S', quindi anche l'insieme S, dovrebbe contenere $\{\chi(x)_{xy}, \chi(x)_{yx}\}$ che non può essere in S per via di come esso è costruito dalla funzione di riduzione E_x.

  3COLtoEXCO-Dimostrazione-Parte2.pdf è il documento di riferimento.

Riduzione EXCO $\leq_P$ Subsetsum 14

14.1 Problema Subsetsum

L'ultimo passo per dimostrare che SAT $\leq_P$ KP è la riduzione EXCO $\leq_P$ Subsetsum in cui EXCO è il problema "*EXact COver*" e Subsetsum è una delle versioni di KP.

La funzione di riduzione per dimostrare EXCO $\leq_P$ Subsetsum è una funzione E_s che, presa una terna di insiemi $((U, S), S')$, in cui $S \subseteq \mathcal{P}(U)$ e $S' \subseteq S$, restituisce una coppia che può essere un'istanza di Subsetsum, ovvero una coppia costituita da un insieme W di numeri che giocano il ruolo di profitti e pesi, e da un numero C, la capacità. In particolare:

- W contiene rappresentazioni numeriche in una qualche base p degli elementi di S';
- p è legato al numero di occorrenze degli elementi di U negli elementi di S;
- Il valore di C risulta essere quello della serie geometrica $p^0 + p^1 + p^2 + \ldots + p^{m-2} + p^{m-1}$, cioè $\frac{1-p^m}{1-p}$.

14.2 Una prima trasformazione

EXCOtoS3-Funzione-riduzione-idea.mp4 guida alla sintesi della prima versione della funzione di trasformazione da terne di insiemi che *possono* essere istanze di EXCO a coppie (insieme, valore) che possono essere istanze di Subsetsum.
EXCOtoS3-Funzione-riduzione-idea.pdf è il documento di riferimento.

EXCOtoS3-Funzione-riduzione-versione1.mp4 illustra una prima versione di E_s che ricava istanze di Subsetsum, partendo da istanze di EXCO. Nella trasformazione gioca un ruolo fondamentale la copertura esatta S' di S. Ogni elemento di S' è codificabile come numero. La somma di tutti i numeri che codificano gli elementi di S' indicano la "capacità" della somma da ottenere espressa come numero binario

$(\underbrace{11\ldots1}_{|U|})_2 = (2^{|U|}-1)_{10}$, dove $|U|$ è la cardinalità di U, $(\underbrace{11\ldots1}_{|U|})_2$ un numero in base 2 e $(2^{|U|}-1)_{10}$ un numero in base 10.
EXCOtoS3-Funzione-riduzione-versione1.pdf è il documento associato.

14.3 Problema della prima trasformazione

EXCOtoS3-Funzione-riduzione-versione1-problema.mp4 presenta una terna $((U,S),X) \notin$ EXCO tale che $E_s((U,S),X) \in$ Subsetsum. Cioè, almeno per la terna $((U,S),X)$ data, non è vero che $((U,S),X) \notin$ EXCO $\Rightarrow E_s((U,S),X) \notin$ Subsetsum perché la somma delle rappresentazioni numeriche degli elementi in X fornisce il valore cercato, ma X non è una copertura esatta di U.

EXCOtoS3-Funzione-riduzione-versione1-problema.pdf è il documento di riferimento.

14.4 Funzione di riduzione

EXCOtoS3-Funzione-riduzione-definitiva.mp4 illustra come risolvere il problema illustrato. Si tratta di scegliere opportunamente la base per rappresentare come numeri gli elementi del secondo argomento S della trasformazione E_s, quindi, di conseguenza, anche degli elementi nel terzo argomento problematico X di E_s. Se X ha forma:

$$\begin{array}{ccccc} x_{1n} & \ldots & x_{1i} & \ldots & x_{11} \\ x_{2n} & \ldots & x_{2i} & \ldots & x_{21} \\ & & \vdots & & \\ x_{mn} & \ldots & x_{mi} & \ldots & x_{m1} \end{array}$$

(notare che gli indici sono elencati in modo da vedere ogni x_{ij} come cifra in un numero la cui cifra meno significativa è a destra) e se abbiamo $m > 2$ ed usiamo 2 come base, è molto plausibile avere la situazione:

$$\begin{array}{ccccc} x_{1n}+ & \ldots & x_{1i}+ & \ldots & x_{11}+ \\ x_{2n}+ & \ldots & x_{2i}+ & \ldots & x_{21}+ \\ & & \vdots & & \\ \hline x_{mn}= & \ldots & x_{mi}= & \ldots & x_{m1}= \\ \hline 1 & \ldots & 1 & \ldots & 1 \end{array},$$

in cui ogni 1 al di sotto di ciascuna colonna risulta dal sommare gli elementi della colonna sotto la quale esso si trova, e del riporto che arriva dalle somme degli elementi nelle colonne alla sua destra:

> Potremmo avere l'impressione che X è una copertura, quando, in realtà, non lo è perché la mancanza di alcuni elementi di U dai sottoinsiemi di X può essere nascosta dalla presenza multipla di uno stesso elemento in più insiemi di X.

Si tratta, quindi di evitare riporti affinché gli 1 contino l'effettiva occorrenza di un elemento in ciascuna colonna in cui possiamo organizzare una copertura esatta

$S' \subseteq S$ di U. *È sufficiente contare con numeri la cui base non permette tali riporti.* La base si ricava contando il numero massimo di occorrenze di ogni elemento x di U negli insiemi di S. Il motivo è che S stesso può essere copertura esatta.

EXCOtoS3-Funzione-riduzione-definitiva.pdf è il documento di riferimento.

14.5 Dimostrazione

EXCOtoS3-Dimostrazione.mp4 illustra un "esploso" della dimostrazione de EXCO $\leq_P$ Subsetsum, data molto stringatamente al fondo di EXCOtoS3-Kozen.

EXCOtoS3-Dimostrazione.pdf è il documento di riferimento.

Visione d'insieme 15

Non c'è niente di più pratico di una buona teoria.

Kurt Lewin

In retrospettiva, questo corso è stato pensato per ricostruire una situazione simile a quella descritta in [GJ79, Sezione. 1.1], definita come "capricciosamente inventiva" e riportata in Garey&Johnson-bandersnatch.pdf

Il nostro "bandersnatch" è stato rappresentato dai problemi Valutazione, Bandi e Rischio di cui sono state date istanze precise, ma a solo titolo d'esempio. L'idea implicita è che se, di Valutazioni, viene data una specifica istanza i, per un informatico è naturale pensare che il programma richiesto per risolvere Valutazioni deve fornire una risposta per l'istanza i e per tutte le altre possibili istanze di Valutazioni, indipendentemente dal numero di coppie (voto massimo per domanda d_i, valutazione della domanda d_i) e dal voto massimo erogabile dall'intera prova.

Le proposte iniziali per dare una risposta ad almeno Valutazioni, forse il problema più immediato da cogliere tra i tre motivazionali proposti, sono state lentamente assorbite dalla strategia seguita per risolverlo: essa è stata imposta dal sapere che Valutazioni è una versione reale di KP.

Questo corso ed i partecipanti ad esso hanno giocato il ruolo del "*boss*" in [GJ79, Sezione. 1.1], mentre lo scrivente ha dovuto impersonare il *company's chief algorithms designer* che spera disperatamente di non doversi fermare al "*I can't find an efficient algorithm, I guess I'm just too dumb.*"

Fortunatamente una schiera di *famous people*, come: S. Cook, L. Levin, R.M. Karp, G.S. Zeitin, o G.S. Zeitin, ha aiutato scrivente e *chief algorithms designer* nell'evitare il licenziamento almeno per non aver saputo risolvere efficientemente Valutazioni.

L'aiuto si è concretizzato tramite la catena di riduzioni seguente:

$$\begin{aligned} \mathsf{SAT} &\leq_{\mathsf{P}} \mathsf{NNF} \\ \mathsf{NNF} &\leq_{\mathsf{P}} \mathsf{CNF} \\ \mathsf{CNF} &\leq_{\mathsf{P}} \mathsf{3CNF} \\ \mathsf{3CNF} &\leq_{\mathsf{P}} \mathsf{3COL} \\ \mathsf{3COL} &\leq_{\mathsf{P}} \mathsf{EXCO} \\ \mathsf{EXCO} &\leq_{\mathsf{P}} \mathsf{Subsetsum} \\ \mathsf{Subsetsum} &\leq_{\mathsf{P}} \mathsf{KP} \ , \end{aligned} \tag{15.1}$$

basata su strumenti formali sviluppati da discipline come la Logica matematica, la Matematica discreta, l'Informatica teorica; di qui la stringente necessità di rammentare l'aforisma di Kurt Lewin.

Grazie alla transitività della relazione $\leq_{\mathsf{P}}$, la catena (15.1) implica:

$$\mathsf{SAT} \leq_{\mathsf{P}} \mathsf{KP} \ , \tag{15.2}$$

ovvero che un algoritmo veloce su ogni istanza di KP è piuttosto difficile da trovare, anche da parte di super esperti. Il motivo è duplice e più forte di quel che si può ricavare guardando la sola riduzione in (15.2) e che (ri)leggiamo come segue ancora una volta:

> La riduzione (15.2) dimostra che KP "contiene" SAT.
>
> Ovvero, se si potesse risolvere velocemente KP su tutte le sue istanze, avremmo un buon algoritmo generale anche per SAT.
>
> Il motivo è che, data una formula proposizionale x, potremmo sapere se $x \in \mathsf{SAT}$, applicando la funzione di riduzione adatta:
>
> $$F : \text{Formule proposizionali} \longrightarrow (\mathbb{N}^n \times \mathbb{N}^n \times \mathbb{N})$$
>
> a x, per poi restituire $\mathcal{K}(F(x))$, in cui $\mathcal{K}$ è l'algoritmo, veloce su tutte le istanze, che risolve KP. Se $\mathcal{K}(F(x)) =$ "Sì", allora $x \in \mathsf{SAT}$. Se $\mathcal{K}(F(x)) =$ "No", allora $x \notin \mathsf{SAT}$.

Il quadro completo, però, si ha rifacendoci al Teorema di Cook-Levin, una delle cui conseguenze è l'esistenza della riduzione opposta a (15.2):

$$\mathsf{KP} \leq_{\mathsf{P}} \mathsf{SAT} \ , \tag{15.3}$$

da cui si ricava che SAT "contiene" KP.

Quindi, KP e SAT, contenendosi a vicenda, sono computazionalmente equivalenti dal punto di vista della difficoltà di trovare un buon algoritmo generale. Siccome SAT è uno dei problemi per cui tenacia ed intensità con le quali è stato cercato un algoritmo generale, sinora senza successo, sono difficilmente quantificabili, pare sia difficile trovarlo per KP; algoritmi per Valutazioni, mera rilettura di KP, condividono lo stesso destino.

Mi piace concludere questa prima parte della rilettura a ritroso del corso, con un compito ideale, in due punti, in carico ad ogni serio *company's chief algorithms designer*:

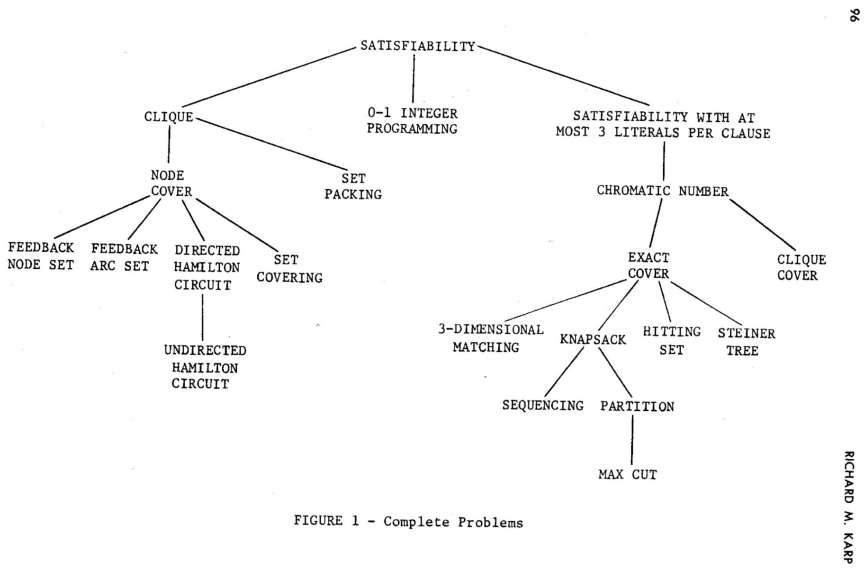

FIGURE 1 - Complete Problems

FIGURA 15.1: Albero di riduzioni tra problemi NP-complete.

- Saper fornire sia istanze formali, sia interpretazioni concrete della maggior parte di problemi noti nella classe dei problemi NP-complete, partendo, ad esempio, dalla lista statica e datata in [GJ79, Appendice A], per poi concentrarsi su una lista più ampia e dinamica come [Wik];
- Conoscere la maggior parte dei dettagli delle riduzioni da un problema all'altro delle quali un albero capostipite è in [Kar10], riedizione de [Kar72], riportato in FIGURA 15.1.

Nel tempo, i ricercatori hanno seguito il principio filosofico noto come "Un colpo al cerchio ed uno alla botte":

- da un lato sono proseguiti gli studi su problemi ritenuti più significativi di altri, ad esempio SAT, 3COL, TSP (commesso viaggiatore), KP, per scovare l'origine della loro difficoltà intrinseca;
- dall'altro si sono cercate tecniche algoritmiche in grado di poter risolvere tali problemi perché è troppo utile saperlo fare per attività di routine e perché, in generale, grazie a studi e pratica, è evidente che le istanze "cigno nero" [Tal09], occorrono raramente.

Il secondo punto giustifica la prima parte del corso. Ogni problema computazionale è configurabile come ricerca. Ci alleniamo a sintetizzare algoritmi individuando le

caratteristiche (proprietà invarianti?) che permettono di scartare le non soluzioni, per convergere verso le soluzioni. Intuitivamente, se:

1. la verifica delle caratteristiche è sempre "locale", cioè dipende sempre da una piccola quantità di lavoro nota a priori,

2. la dimensione dell'insieme delle non soluzioni scartate ad ogni passo della ricerca "è sempre cospicuo",

allora si può convergere "velocemente" verso risposte ad un problema. Per problemi come SAT, KP, etc. almeno uno tra i punti 1 e 2 non è evidente a nessuno come possa essere vero. Quindi, si procede alla sintesi di algoritmi per approssimazioni successive. Di qui le tappe naturali seguite dalla prima parte el corso:

Brute-Force. In assenza di proprietà invarianti evidenti, che possano caratterizzare velocemente insiemi sempre più piccoli di configurazioni entro cui cercare, o non cercare, una soluzione, si elencano tutte le alternative, fino ad una prima soluzione trovata. In generale, elencare esaustivamente suggerisce primi criteri, a basso costo computazionale, per individuare insiemi di non soluzioni.

Backtrack. È *Brute-Force* esteso a sfruttare i criteri a basso costo di cui sopra, detti di *pruning*, per non verificare, caso per caso, tutte le componenti di un insieme di configurazioni per le quali sia evidente che non possono contenere risposta alcuna.

I criteri di *pruning* possono essere più o meno efficaci, riguardo alla dimensione dell'insieme di non soluzioni tagliate.

Branch&Bound. È un *Backtrack* in cui:

1. l'applicazione dei criteri di *pruning* è svincolata dalla struttura dello spazio di configurazioni in cui cercare le soluzioni;
2. i criteri di *pruning* forniscono valutazioni quantitative riguardo alla possibilità di contenere una soluzione da parte di un sottoinsieme di configurazioni.

   La valutazione quantitativa esprime la qualità della risposta che un sottoinsieme di configurazioni deve assicurare.

   Un insieme di configurazioni è scartato perché contiene solo non risposte o soluzioni non migliori di quelle note, ovvero non assicura il livello di qualità richiesto.

KP ha caratteristiche per cui si individua un criterio di *pruning* molto efficace. Il suo rilassamento lineare LKP gode della *Greedy Optimality Property*: riempire il sacco fino all'orlo, usando una porzione dell'ultimo *item* che vi entra, avendo cura di massimizzare il profitto per unità di misura del peso (o volume) ad ogni scelta di *item*, assicura il maggior profitto.

Riassumendo, abbiamo realizzato un'istanza dello schema in Figure 15.2 in cui P e C sono parametri:

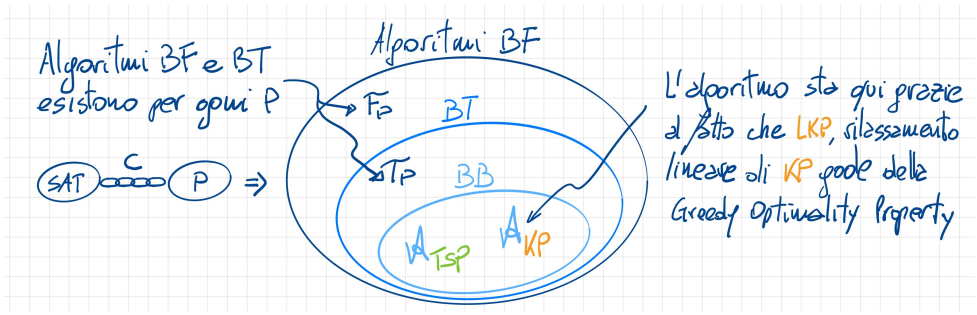

FIGURA 15.2: Schema metodologico seguito.

P è un problema intrinsecamente difficile, per il quale, per gradi successivi, possiamo anche produrre un algoritmo *Branch&Bound*, se le sue caratteristiche lo permettono;

C è la catena di riduzioni che dimostra la difficoltà intrinseca di P.

In FIGURA 15.2 il problema P rappresenta KP, e la catena C rappresenta (15.1).

È interessante che lo schema rimane valido usando TSP al posto di KP e la seguente catena, al posto di (15.1):

$$\begin{aligned} \mathsf{SAT} &\leq_{\mathsf{P}} \mathsf{CLI} \\ \mathsf{CLI} &\leq_{\mathsf{P}} \mathsf{VEC} \\ \mathsf{VEC} &\leq_{\mathsf{P}} \mathsf{HC} \\ \mathsf{HC} &\leq_{\mathsf{P}} \mathsf{TSP} \ , \end{aligned} \tag{15.4}$$

in cui CLI è il problema "CLIque", VEC è il problema "Vertex Cover" e HC è il problema "Hamiltonian Cicle".

Il bello sta nello scoprire sia altre istanze della metodologia qui riassunta, sia altri schemi in cui al posto della tecnica *Branch&Bound* se ne trovano di equivalentemente interessanti. Una di esse è la riduzione a problemi di minimizzazione identificati come *modelli* QUBO, la cui minimizzazione sembra poter essere accelerata grazie a computer quantistici che computano in accordo con il modello detto *Adiabatic Quantum Computing*.

Parte IV

Modelli quadratici

Adiabatic Quantum Computing 16

Dimostrare che almeno un algoritmo con complessità asintotica non esponenziale nella dimensione dell'input esiste per almeno un problema della classe NP-complete, potrebbe non essere l'unica strada per tentare implementazioni efficienti.

Una possibile via è rivolgersi a modelli computazionali non convenzionali che promettono un incremento sostanziale di quanto è calcolabile nell'unità di tempo di riferimento, rispetto a modelli di computazione classici.

Esistono modelli di computazione basati su operazioni primitive ispirate alla, o che sfruttano proprietà della, meccanica quantistica per abbattere la complessità computazionale di problemi che risultano (sinora) intrattabili da parte di modelli classici equivalenti alle Macchine di Turing.

Intrattabile2Trattabile-GMvsAQC.mp4 giustifica brevemente perché, potendo scegliere tra i due modelli di computazione quantistica oggi "disponibili", ci orientiamo verso il modello *Adiabatic Quantum Computing* (AQC) che funziona in accordo con l'analogia "Computazione come Hamiltoniano".

Un Hamiltoniano è una matrice Hermitiana, cioè una matrice di elementi in $\mathbb{C}$, la classe dei numeri complessi, con specifiche e notevoli proprietà. I riferimenti bibliografici abbondano, ma a noi non interesserà conoscere il dettaglio dell'algebra con cui manipolare Hermitiani o Hamiltoniani. È sufficiente avere l'idea che un Hamiltoniano:

- può essere interpretato come caratterizzazione del livello di energia istantaneo di un sistema fisico;
- può descrivere l'evoluzione dello stato di tale sistema fisico.

Intrattabile2Trattabile-GMvsAQC.pdf è il documento di riferimento.

16.1 Intuizioni su *Adiabatic Quantum Computing*

- Intrattabile2Trattabile-via-AQC.mp4 fornisce una possibile intuizione sul significato di "computare tramite Hamiltoniani" che sta alla base della *Adiabatic Quantum Computation*.

  L'idea principale consiste nel ridurre un problema alla minimizzazione di una funzione rappresentata in forma matriciale, cioè un Hamiltoniano $\mathcal{H}_P$, che è un operatore in uno spazio vettoriale in cui:

  – le configurazioni generate dalle variabili del problema diventano vettori;

  – i valori assunti dalla funzione da minimizzare sono autovalori di $\mathcal{H}_P$ e i vettori che rappresentano le configurazioni generate dalle varibili sono autovettori di $\mathcal{H}_P$.

  È allora possibile inserire $\mathcal{H}_P$ in una opportuna combinazione lineare di Hamiltoniani che descrivono la trasformazione di un registro di *qbit* da uno stato iniziale ad uno finale. Se la transizione "stato iniziale → stato finale" avviene abbastanza lentamente, con alta probabilità lo stato finale contiene configurazioni da cui leggere le risposte al problema iniziale.

  Intrattabile2Trattabile-via-AQC.pdf è il documento di riferimento.

- Intrattabile2Trattabile-PLI2AQC.mp4 illustra l'idea di come procedere per ridurre un problema P in *Programmazione Lineare Intera* ad un'istanza del problema che può essere risolto da Qcpu che realizzano il modello "computazione tramite Hamiltoniano". Il punto sta nel rappresentare P, inclusi i suoi vincoli lineari, in un problema di minimizzazione QUBO (*Quantum Unconstraint Binary Optimization*) o Ising[1]. Un problema in forma QUBO/Ising ha una sua riformulazione naturale a grafo; quest'ultimo va mappato nel grafo fisico tra i *qbit* della Qcpu di riferimento. La mappatura, il cui nome tecnico è *minor-embedding*, preserva le relazioni tra i nodi del grafo QUBO/Ising, ma può far corrispondere un nodo del grafo QUBO/Ising a più nodi del grafo fisico nella Qcpu. Per quanto sperimentalmente funzioni bene, il *minor-embedding* è un problema NP-complete! Questo è un campanello d'allarme sul fatto che voler sfruttare architetture adatte a sviluppare una computazione adiabatica quantistica, cioè che implementano l'analogia "computazione come Hamiltoniano", può non dare i risultati sperati.

  Intrattabile2Trattabile-PLI2AQC.pdf è il documento di riferimento.

16.1.1 Qualche cautela su AQC

- Intrattabile2Trattabile-minor-embedding.mp4 illustra il dettaglio di un esempio tratto da "*Theory versus practice in annealing-based quantum computing*" [McG20] in cui occorre eseguire il *minor-embedding* di un grafo la cui struttura

[1] Scienziato che lo introdusse in relazione a studi del tutto indipendenti dall'ottimizzazione combinatoria

non è immediatamente adattabile a quella del grafo fisico della Qcpu di riferimento. In particolare, occorre mappare un paio di nodi del grafo QUBO/Ising ciascuno in una coppia di nodi del grafo fisico.

Intrattabile2Trattabile-minor-embedding.pdf è il documento di riferimento.

- Intrattabile2Trattabile-Limiti-AQC.mp4 riassume brevemente quelli che potremmo definire "punti critici" del passaggio da algoritmi classici ad algoritmi quantistici basati su AQC. La sorgente principale dei problemi è la difficoltà nel fornire un *lower bound* al tempo di *annealing*, cioè il tempo minimo necessario per passare dallo stato iniziale della computazione tramite Hamiltoniano a quello finale. Il messaggio fondamentale è che, se per un problema classico P il *lower bound* non è noto, allora la forma QUBO che fornisce un algoritmo risolutivo per P basato su *quantum annealing* è paragonabile ad una *euristica*, cioè ad una implementazione algoritmica di cui l'efficienza e la correttezza non sono valutabili in generale.

  I casi in cui il *lower bound* è noto non abbondano.

  Inoltre, ci sono problemi P per i quali all'aumentare della dimensione delle istanze la *performance* di un algoritmo quantistico adiabatico per P degrada maggiormente di quanto non succeda ai corrispettivi algoritmi classici.

  Intrattabile2Trattabile-Limiti-AQC.pdf è il documento di riferimento in cui, per completezza e per i soli eventuali curiosi, lascio i riferimento ad una versione semplificata del teorema adiabatico il quale formalizza l'idea che una computazione tramite Hamiltoniano non può essere "troppo veloce".

16.2 Visione più ampia su *Adiabatic Quantum Computing*

L'intenzione delle sezioni precedenti è rendere ragionevolmente accessibile una parte de "*Theory versus practice in annealing-based quantum computing*" [McG20], panoramica di AQC che riassume lo stato dell'arte, propone una sistematizzazione terminologica, tenta un legame tra i mondi speculativo e sperimentale che gravitano attorno ad AQC, ed alla sua declinazione "*implementation-oriented*" identificata come *Quantum Annealing* (QA), attraverso l'analogia:

$$\frac{\text{AQC}}{\text{Turing Machines}} \simeq \frac{\text{QA}}{\text{RAM}}$$

Un possibile completamento sono "Introduzione e bibliografia de *Adiabatic Quantum Computing*", estratto da [AL18].

Globalmente, da entrambe le letture suggerite si ricava l'impressione di un campo dell'informatica ancora agli inizi e con un potenziale di sviluppo amplissimo.

Guida per i contenuti di questo capitolo è [GKD19].

17.1 Problema QUBO

QUBO-triangolare-sup.mp4 introduce l'essenziale sui *Quadratic Unconstrained Binary Models* (QUBO).

Definizione 17.1.1 (Problema QUBO) Un problema QUBO richiede di trovare un'istanza $x^* = (x_1^*, \ldots, x_n^*)$ delle variabili $x_1, \ldots, x_n$ in grado di:

$$\begin{aligned} &\text{minimizzare} \quad \overline{x}^t Q \overline{x} \\ &x_1, \ldots, x_n \in \{0, 1\} \end{aligned} \tag{17.1}$$

in cui $\overline{x}$ è il vettore $\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$, il vettore $\overline{x}^t$ è il trasposto di $\overline{x}$ e Q è una matrice *triangolare superiore* (questo vincolo può essere rilassato). □

Esempio 17.1.1 (Istanza QUBO) L'istanza QUBO:

$$\text{minimizzare} \quad \underbrace{[x_1 \ x_2 \ x_3 \ x_4]}_{\overline{x}^t} \underbrace{\begin{bmatrix} -5 & 4 & 8 & 0 \\ 0 & -3 & 4 & 0 \\ 0 & 0 & -8 & 10 \\ 0 & 0 & 0 & -6 \end{bmatrix}}_{Q} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\overline{x}} . \tag{17.2}$$

è caratterizzata da una matrice triangolare superiore Q.

Trovare almeno una risposta per (17.2), cioè una quadrupla che minimizza (17.2), equivale a trovarla per il problema:

$$\begin{aligned} \text{minimizzare} \ & -5x_1^2 - 3x_2^2 - 8x_3^2 - 6x_4^2 \\ & + 4x_1x_2 + 8x_1x_3 + 4x_2x_3 + 10x_3x_4 \end{aligned} \tag{17.3}$$

che, però, viene preso nella seguente forma *semplificata*:

$$\begin{aligned} \text{minimizzare } & -5x_1 - 3x_2 - 8x_3 - 6x_4 \\ & + 4x_1x_2 + 8x_1x_3 + 4x_2x_3 + 10x_3x_4 \end{aligned} \tag{17.4}$$

cioè, nella forma in cui le potenze $x_1^2, x_2^2, x_3^2, x_4^2$ sono sostituite dai termini lineari x_1, x_2, x_3, x_4, rispettivamente, perché $x_1 = x_1^2, x_2 = x_2^2, x_3 = x_3^2, x_4 = x_4^2$ quando, come nel nostro caso, le variabili sono limitate al dominio $\{0, 1\}$.

Vale la pena osservare che:

- i coefficienti delle occorrenze lineari x_1, x_2, x_3, x_4 costituiscono la diagonale principale di Q;
- ogni coefficiente Q_{ij} al di sopra della diagonale principale esprime il grado di "influenza reciproca" tra le variabili x_i e x_j. □

QUBO-triangolare-sup.pdf è il documento di riferimento.

17.1.1 Modello QUBO in forma simmetrica

La forma della matrice Q che identifica un problema QUBO può essere rilassata. Sono ammesse anche matrici simmetriche rispetto alla diagonale principale, come conseguenza di semplici relazioni esistenti tra matrici triangolari superiori e matrici simmetriche.

QUBO-simmetrico.mp4 dà l'intuizione del perché il problema QUBO (17.2) (o (17.3), o (17.4)) ha le stesse soluzioni di:

$$\text{minimizzare } \underbrace{[x_1\ x_2\ x_3\ x_4]}_{\overline{x}^t} \underbrace{\begin{bmatrix} -5 & 2 & 4 & 0 \\ 2 & -3 & 2 & 0 \\ 4 & 2 & -8 & 5 \\ 0 & 0 & 5 & -6 \end{bmatrix}}_{Q^s} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\overline{x}} . \tag{17.5}$$

In (17.5) la matrice Q^s è simmetrica ed ottenuta da Q in (17.2) come segue:

- Q^s_{ij} e Q^s_{ji} è identico a $\frac{Q_{ij}}{2}$, per ogni $1 \leq i < j \leq 4$;
- $Q^s_{ii} = Q_{ii}$, per ogni $1 \leq i \leq 4$.

QUBO-simmetrico.pdf è il documento di riferimento.

17.1.2 Modelli QUBO *vs.* modelli Ising

Un problema Ising ha la stessa forma sintattica di un problema QUBO, ma le variabili possono assumere i valori interi dell'insieme $\{1, -1\}$, non dell'insieme $\{0, 1\}$.

Definizione 17.1.2 (Problema Ising) Un problema Ising richiede di trovare un'istanza $s^* = (s_1^*, \ldots, s_n^*)$ delle variabili $s_1, \ldots, s_n$ in grado di:

$$\begin{aligned} &\text{minimizzare} \quad \overline{s}^t J \overline{s} \\ &s_1, \ldots, s_n \in \{1, -1\} \end{aligned} \tag{17.6}$$

in cui $\overline{s}$ è il vettore $\begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix}$, il vettore $\overline{s}^t$ è il trasposto di $\overline{s}$ e J una matrice triangolare superiore o, indifferentemente, simmetrica rispetto alla diagonale principale. □

Esempio 17.1.2 (QUBO *vs.* Ising) QUBOvsIsing.mp4 illustra la relazione tra il problema QUBO (17.5) e il problema Ising seguente:

$$\text{minimizzare} \quad \underbrace{\begin{bmatrix} s_1 & s_2 & s_3 & s_4 \end{bmatrix}}_{\overline{s}^t} \underbrace{\begin{bmatrix} -\frac{1}{2} & \frac{1}{8} & \frac{5}{8} & 0 \\ \frac{1}{8} & -\frac{1}{2} & \frac{3}{8} & 0 \\ \frac{5}{8} & \frac{3}{8} & -2 & \frac{5}{4} \\ 0 & 0 & \frac{5}{4} & -\frac{1}{2} \end{bmatrix}}_{J} \underbrace{\begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix}}_{\overline{s}}, \tag{17.7}$$

sottolineando alcuni aspetti rilevanti nel passare da una all'altro. Si sottolinea anche che, nel passaggio, l'insieme delle soluzioni cambia siccome i domini dei due problemi cui le variabili appartengono sono diversi; tuttavia gli insiemi delle risposte sono isomorfi, cioè ad una soluzione del problema in forma QUBO corrisponde esattamente una soluzione della forma Ising.

QUBOvsIsing.pdf è il documento di riferimento. □

17.1.3 Da Ising a Qcpu

Ising2Qcpu.mp4 illustra come la struttura di una istanza Ising corrisponde ad un grafo il quale, a sua volta, può essere associato al sotto-grafo fisico che costituisce la rete di *qbit* della Qcpu di riferimento, che sviluppa una computazione tramite Hamiltoniano. Si sottolinea la possibilità di costruire *catene* di *qbit* fisici della Qcpu che rappresentano una singola variabile del problema Ising, cioè un nodo del grafo corrispondente. Nel caso in esame le catene sono costituite da:

- coppie di *qbit* fisici cui sono assegnati pesi dimezzati rispetto a quelli delle variabili corrispondenti;
- un singolo arco tra tali coppie di *qbit* fisici. Il peso dell'arco assume valore "logico" $-\infty$, indicando che si cerca di imporre alla Qcpu il vincolo che i due *qbit* legati in tal modo dovranno assumere lo stesso valore nello stato finale.

Ising2Qcpu.pdf è il documento di riferimento.

17.2 D-Wave API/IDE per BQM (QUBO/Ising)

Usiamo API/IDE disponibili una volta ottenuto un account su D-Wave Leap per definire i *Binary Quadratic Models* (BQM) descritti per mezzo degli esempi sviluppati in precedenza, seguendo [GKD19]; in accordo con la terminologia D-Wave, BQM indica indifferentemente modelli QUBO o Ising.

- GloverKD19Matrice_ESSA.mp4 illustra l'implementazione dell'**Esempio 17.1.1**, esprimendo la matrice Q in (17.2) nel modo ovvio, come *dictionary* Python, sfruttando l'API che, date anche le variabili, la trasforma in un BQM.

  In questo caso l'unica risposta $(x_0^* = 1, x_1^* = 0, x_2^* = 0, x_3^* = 1)$ è individuata in due modi:

  - per mezzo del "campionatore" `ExactSolver` che, in realtà, effettua una visita *Brute-Force* di tutto lo spazio degli stati;
  - per mezzo di un vero campionatore, chiamato `SimulatedAnnealingSampler`, implementazione dell'algoritmo Simulated Annealing - Wikypedia.

  `GloverKD19Matrice_ESSA.py` è il codice di riferimento per entrambe le soluzioni.

- GloverKD19Polinomio_ESSA.mp4 illustra l'implementazione dell'**Esempio 17.1.1**, esprimendo il polinomio in (17.4) nel modo ovvio, come una espressione Python *standard*, sfruttando l'API che lo trasforma in un BQM.

  Anche in questo caso, la risposta $(x_0^* = 1, x_1^* = 0, x_2^* = 0, x_3^* = 1)$ è determinata con i due campionatori disponibili.

  `GloverKD19Polinomio_ESSA.py` è il codice di riferimento.

18.1 Formulazioni QUBO naturali

Guida per i contenuti di questo capitolo è [GKD19].

18.1.1 "Sotto-insiemi a Somma Identica" $\leq_{\mathsf{P}}$ QUBO

NumberPartitioning.mp4 riassume quanto segue, cioè come ridurre a forma QUBO il problema che, dato un insieme numerico S, vi cerca un sottoinsieme T i cui elementi, una volta sommati, diano un valore identico alla somma degli elementi in $S \setminus T$.

Definizione 18.1.1 Supponiamo sia dato un insieme $S = \{v_1, \ldots, v_n\}$ di numeri naturali. Se possibile, trovare un sotto-insieme $T \subset S$ tale che $\sum_{v \in T} v = \sum_{v \in S \setminus T} v$. □

La prima osservazione è che vale l'eguaglianza:

$$\sum_{v \in S \setminus T} v = \sum_{v \in S} v - \sum_{v \in T} v \ . \tag{18.1}$$

Grazie a (18.1), il problema si riduce a trovare $T \subset S$ tale che:

$$\left(\sum_{v \in T} v = \sum_{v \in S \setminus T} v = \sum_{v \in S} v - \sum_{v \in T} v \right) \Leftrightarrow \left(0 = \sum_{v \in S} v - 2 * \sum_{v \in T} v \right) \ . \tag{18.2}$$

L'equazione (18.2) suggerisce di trovare T, risolvendo il seguente problema che si trova implicitamente in forma QUBO:

$$\text{minimizzare} \quad \left(\sum_{v \in S} v - 2 * \sum_{i \in \{1,\ldots,n\}} x_i v_i \right)^2 \ . \tag{18.3}$$

18.1.1.1 Prima istanza

Prendiamo l'insieme $S = \{v_1 = 1, v_2 = 3, v_3 = 4, v_4 = 2, v_5 = 6\}$. Svolgendo (18.3), si ricava la matrice Q:

$$\begin{bmatrix} -60 & 24 & 32 & 16 & 48 \\ 0 & -156 & 96 & 48 & 144 \\ 0 & 0 & -192 & 64 & 192 \\ 0 & 0 & 0 & -112 & 96 \\ 0 & 0 & 0 & 0 & -240 \end{bmatrix}$$

mentre il polinomio esplicito è:

$$\begin{aligned} 0 = & -x1 * 60 + x2 * x1 * 24 + x3 * x1 * 32 + x4 * x1 * 16 + x5 * x1 * 48 \\ & - x2 * 156 + x3 * x2 * 96 + x4 * x2 * 48 + x5 * x2 * 144 \\ & - x3 * 192 + x4 * x3 * 64 + x5 * x3 * 192 \\ & - x4 * 112 + x5 * x4 * 96 \\ & - x5 * 240 \\ & + 256.0 \end{aligned}$$

nel quale compare l'*offset* 256.

L'istanza S data ha le due risposte $(x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 1, x_5 = 1)$ e $(x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 0)$.

18.1.1.2 Seconda istanza

Prendiamo l'insieme $S = \{v_1 = 1, v_2 = 1, v_3 = 1, v_4 = 1, v_5 = 1\}$. Svolgendo (18.3), si ricava la matrice Q:

$$\begin{bmatrix} -16 & 8 & 8 & 8 & 8 \\ 0 & -16 & 8 & 8 & 8 \\ 0 & 0 & -16 & 8 & 8 \\ 0 & 0 & 0 & -16 & 8 \\ 0 & 0 & 0 & 0 & -16 \end{bmatrix}$$

mentre il polinomio esplicito è:

$$\begin{aligned} 0 = & - x1 * 16 + x2 * x1 * 8 + x3 * x1 * 8 + x4 * x1 * 8 + x5 * x1 * 8 \\ & - x2 * 16 + x3 * x2 * 8 + x4 * x2 * 8 + x5 * x2 * 8 \\ & - x3 * 16 + x4 * x3 * 8 + x5 * x3 * 8 \\ & - x4 * 16 + x5 * x4 * 8 \\ & - x5 * 16 \\ & + 25 \end{aligned}$$

nel quale compare l'*offset* 25. In questo caso non possiamo avere risposte.

NumberPartitioning.pdf è il documento di riferimento.

`NumberPartitioning.py` è il sorgente che campiona lo spazio degli stati relativi a due istanze di S presentate nella sottosezioni 18.1.1.1 e 18.1.1.2 seguenti.

18.1.2 MAXCUT $\leq_P$ QUBO

MaxCut2QUBO.mp4 ricorda il problema MAXCUT e descrive su un semplicissimo esempio, il principio secondo cui la riduzione MAXCUT $\leq_P$ QUBO funziona.

L'idea essenziale è tentare di massimizzare il conteggio degli archi tra nodi che appartengono ai due insiemi in cui il grafo dato debba essere "tagliato". Questo equivale a:

- assegnare "1" ogni arco (i, j) pr cui i è in un insieme e j nell'altro;
- cercare una partizione dei nodi che massimizzi la somma degli archi di peso "1".

Siccome un problema in forma QUBO esprime la necessità di minimizzare un polinomio, si tratterà di minimizzare la negazione della somma di archi di peso "1".

MaxCut2QUBO.pdf e `MaxCut_SA.py` sono documento e codice sorgente associati.

`MaxCutGlover_SA.py` è il sorgente che sviluppa l'esempio in [GKD19].

18.1.2.1 Esercizio

La riduzione descritta qui sopra interpreta ogni arco (i, j) come se fosse una disgiunzione esclusiva, assegnando il peso "1" quando i e j sono in "partizioni" distinte del grafo.

Verificare che esiste un'alternativa, assegnando *energia* "0" ad ogni arco (i, j) con i e j in "partizioni" distinte, cioè interpretando ogni arco come se fosse una bi-implicazione.

18.2 Da problemi con vincoli standard a QUBO

18.2.1 Tipici vincoli come polinomio penalità

Vincoli2Penalita.mp4 illustra un metodo standard per trasformare vincoli tipici e semplici di problemi combinatori in polinomi quadratici multivariati che possono essere usti come penalità da sommare ad un Hamiltoniano da minimizzare.

Vincoli2Penalita.pdf è il documento associato.

Note 18.2.1 Nel documento si parla di "Hamiltoniano per Nand" pur essendo esso associato alla tabella di valori di verità dell'operatore logico And.

La giustificazione al nome deriva dal fatto che usiamo un Hamiltoniano per identificare configurazioni a minima energia. Se chiamiamo $\mathcal{H}_{\mathsf{Nand}}$ l'Hamiltoniano per Nand, esso è progettato per soddisfare la proprietà:

$$\mathcal{H}_{\mathsf{Nand}}(x, y) = 0 \Leftrightarrow \mathsf{Nand\ x\ y} = 1 \ .$$

Identiche giustificazioni valgono per gli altri Hamiltoniani trattati, che possiamo identificare come $\mathcal{H}_{\mathsf{Sse}}, \mathcal{H}_{\mathsf{Se}}, \mathcal{H}_{\mathsf{Xor}}$ e $\mathcal{H}_{\mathsf{Or}}$. □

18.2.2 MVC $\leq_P$ QUBO

Indichiamo con MVC il problema *minimum vertex cover*. La riduzione MVC $\leq_P$ QUBO è un esempio naturale di situazione in cui si presenta la necessità di trasformare un vincolo tra quelli illustrati nella sezione 18.2.1 in un Hamiltoniano penalità da sommare ad un Hamiltoniano obiettivo da minimizzare.

Definizione 18.2.1 (Minumum Vertex Cover (MVC)) Sia dato un grafo $G = (V, E)$ non diretto. Si dice che $C \subseteq V$ è un *vertex cover* di G se e solo se per ogni $(i, j) \in E$ si ha che $i \in C$ o $j \in C$. Il *vertex cover* C è *minimo* se, eliminando da C un qualsiasi suo elemento, esiste un qualche arco di E che non è più incidente in alcuno dei vertici di C. □

(Per una disattenzione nel presentare lo schermo del tablet . . .) MVC2MinimizzazioneQUBO-01.mp4 (significativo sino al minuto 4:22) e MVC2MinimizzazioneQUBO-02.mp4 (completa il precedente) illustrano i contenuti delle due sottosezioni 18.2.2.1 e 18.2.2.2 col dettaglio su come si modella il problema MVC prima come problema di minimizzazione, poi come modello QUBO.

18.2.2.1 MVC come minimizzazione

Proprietà 18.2.1 (MVC come minimizzazione) Dato $G = (V, E)$, l'insieme $\{c_1, \ldots, c_n\} = C \subseteq V$ è una copertura minima di G se e solo se C è una risposta del problema:

$$\begin{array}{lll} \text{minimizzare } \operatorname{mvc}(v_1, \ldots, v_n) & & \\ v_i + v_j \geq 1 & \text{per ogni } (i,j) \in E & (18.4) \\ v_i \in \{0, 1\} & \text{per ogni } i \in V & \end{array}$$

in cui $\operatorname{mvc}(v_1, \ldots, v_n) = \sum_{i=1}^{|V|} v_i$. □

La proprietà è giustificata dalle seguenti osservazioni:

- Per ogni vertice $i \in V$, definiamo una variabile v_i come segue:

$$v_i = \begin{cases} 1 & \text{se } i \in C \\ 0 & \text{se } i \notin C \end{cases} . \tag{18.5}$$

  È quindi possibile contare il numero di elementi nella copertura C relativa a G per mezzo dell'ovvio polinomio lineare seguente:

$$\operatorname{mvc}(v_1, \ldots, v_n) = \sum_{i=1}^{|V|} v_i$$

  in cui $|V|$ è la cardinalità di V. Osserviamo che un polinomio lineare è anche quadratico.

- Dato un qualsiasi nodo $i \in V$, se esso è in una copertura minima di C, allora $v_i = 1$.

  Quindi, per ogni arco $(i, j) \in E$, deve valere:

  $$v_i + v_j \geq 1$$

  affinché almeno uno tra i, j sia in una copertura C. Infatti, se per $(i, j) \in E$ avessimo $v_i + v_j = 0$, nessuno dei due potrebbe appartenere ad una copertura.

18.2.2.2 MVC come QUBO

Proprietà 18.2.2 (MVC in forma QUBO) Dato $G = (V, E)$, l'insieme $\{c_1, \ldots, c_n\} = C \subseteq V$ è una copertura minima di G se e solo se C è una risposta del problema:

$$\text{minimizzare} \left(\text{mvc}(v_1, \ldots, v_n) + \lambda * \sum_{(i,j)\in E} (1 - v_i - v_j + v_i v_j) \right) \tag{18.6}$$

per λ (parametro *lagrangiano*) sufficientemente grande. □

Dato $G = (V, E)$, la proprietà è giustificata dalle seguenti osservazioni:

1. il valore di $\text{mvc}(v_1, \ldots, v_n)$ è tanto minore quanto minore è il numero di vertici nella copertura C;

2. Ogni polinomio $1 - v_i - v_j + v_i v_j$ della sommatoria $\sum_{(i,j)\in E}(1 - v_i - v_j + v_i v_j)$ non concorrerà ad incrementare il valore finale di (18.6) quando $c_i + c_j \geq 1$ è soddisfatto, cioè quando almeno uno dei due vertici i e j di (i, j) è coperto, siccome, in quel caso, $1 - v_i - v_j + v_i v_j = 0$.

   È come dire che il valore di (18.6) è minimo quando la prima componente $\text{mvc}(v_1, \ldots, v_n)$ non cresce "troppo", contando vertici, mentre la seconda componente $\sum_{(i,j)\in E}(1 - v_i - v_j + v_i v_j)$ non "riesce" a contare l'esistenza di archi scoperti.

MVC2MinimizzazioneQUBO.pdf è il documento associato ad entrambe le sottosezioni 18.2.2.1 e 18.2.2.2.

18.2.2.3 Importanza del lagrangiano in "MVC come QUBO"

MVC-Lagrangiano.mp4 illustra l'importanza ricoperta dal valore scelto per il lagrangiano nel semplice esempio di riferimento sviluppato in precedenza. In particolare, un lagrangiano appena superiore a 1 penalizza maggiormente, e correttamente, un arco non coperto rispetto all'assenza dalla copertura di un qualche vertice: il lagrangiano raddoppia il costo di ogni arco scoperto di cui si tiene conto nella parte di (18.8) che codifica i vincoli lineari, cioè $2 * \sum_{(i,j)\in E}(1 - v_i - v_j + v_i v_j)$.

- `MVC_lagrangiano_errato_ES.py` permette di sperimentare l'effetto di un lagrangiano troppo piccolo, in questo caso pari a 1. Esso permette di individuare come risposte coperture che non sono minime del grafo:

$$\begin{aligned} V &= \{1,2,3,4,5\} \\ E &= \{(1,2),(2,3),(3,4),(4,1),(3,5),(4,5)\} \ . \end{aligned} \tag{18.7}$$

Il motivo è che un arco (i,j) non coperto, cioè con i e j esclusi dalla copertura, aggiunge 1 alla somma finale (18.6), eventualmente bilanciando un 1 che, invece, è escluso dal conteggio in $\text{mvc}(c_1,\ldots,c_n)$.

Le tuple nella seguente tabella:

v_1	v_2	v_3	v_4	v_5	(18.6)
0	0	1	1	0	3
0	1	0	1	0	3
1	0	1	0	0	3

sono quelle per cui la visita *Brute-Force* produce il valore energetico minimo 3, in relazione al grafo (18.7). Tuttavia esse includono due soli vertici che sono ovviamente insufficienti a determinare una copertura: siamo di fronte a non risposte.

- `MVC_lagrangiano_corretto_ES.py` permette di sperimentare l'effetto di un lagrangiano pari ad almeno 2, che genera la seguente istanza:

$$\text{minimizzare } \text{mvc}(v_1,\ldots,v_n) + 2 * \sum_{(i,j)\in E} (1 - v_i - v_j + v_i v_j) \tag{18.8}$$

di (18.6), la quale fornisce tutte e sole le risposte cercate.

Per completezza, `MVC_lagrangiano_corretto_SA.py` è una versione dei sorgenti precedenti che usa un campionatore *Simulated Annealing*, producendo 4 potenziali risposte, a seguito di un 20 applicazioni del campionatore.

MVC2QUBO-lagrangiano.pdf è il documento di riferimento.

18.2.3 Max2SAT $\leq_P$ QUBO

Max2Sat2QUBO-Problema.mp4 ricorda, aggiungendo un semplice esempio, le seguenti formalizzazioni possibili del problema Max2SAT.

Definizione 18.2.2 (Problema Max2SAT) Sia data una formula proposizionale $f \in \mathsf{pf}$ costituita da una congiunzione di clausole ciascuna con almeno un letterale, ma non più di due.

- ***Versione decisionale.*** Dato $k \in \mathbb{N}$, esiste un'assegnazione di valori di verità alle variabili in $\text{FV}(f)$ che soddisfa (rende vere) almeno k clausole di f?
- ***Versione decisionale.*** Trovare un'assegnazione di valori di verità ϕ^*, ottimale, cioè che soddisfa (rende vere) il maggior numero possibile di clausole di f. □

Max2Sat2QUBO-Problema.pdf è il documento di riferimento.

Max2Sat2QUBO-Riduzione.mp4 introduce quanto segue:

- una funzione F che, presa una formula f costituita da sole congiunzioni di clausole con almeno un letterale, ma non più di due, produce dei polinomi quadratici multivariati con le variabili in $\mathrm{FV}(f)$;
- un esempio che, estratto il polinomio $F(f)$, illustra che al numero massimo di clausole soddisfatte corrisponde il minimo valore $F(f)$.

Max2Sat2QUBO-Riduzione.pdf è il documento di riferimento.

Max2Sat2QUBO-Proprietà.mp4 illustra che, data una formula f, fissando un'assegnazione di valori di verità ϕ per f, su ogni singola clausola C in f, si ha $\phi(C) = 1 - \phi(F(f))$. Da questa osservazione segue che ricavare il numero di clausole di f soddisfatte da ϕ si riduce a contare il numero di clausole per cui $\phi(F(f)) = 0$. Pertanto, il massimo numero di clausole di f soddisfatte da ϕ dipende dal minimo numero di clausole per cui $\phi(F(f)) = 0$, che è un problema in forma QUBO.

Max2Sat2QUBO-Proprietà.pdf è il documento di riferimento.

Infine, `Max2SAT.py` è il sorgente che permette qualche sperimentazione con f e $F(f)$ di riferimento.

Note 18.2.2 È rilevante ed interessante osservare che la dimensione della matrice che costituisce il modello QUBO cui si riduce una problema di soddisfacibilità dipende dal numero di variabili e non dal numero di clausole: l'Hamiltoniano corrispondente ha tanti polinomi quante sono le interazioni non nulle tra coppie di variabili. Sono i fattori moltiplicativi dei vari polinomi che lo compongono a dipendere dal numero di clausole. □

18.3 Da generico problema in PLI a QUBO

Illustriamo un intero processo di riduzione da un problema P in Programmazione Lineare Intera (PLI) di minimizzazione ad un problema equivalente in forma QUBO.

Definizione 18.3.1 P è un problema in PLI se richiede di minimizzare, o massimizzare, un polinomio multivariato lineare $f(x_1, \ldots, x_n)$, simultaneamente rispettando un insieme di vincoli, ciascuno dei quali ha forma:

$$C(x_1, \ldots, x_n) \mathcal{R} v \tag{18.9}$$

in cui:

- $\mathcal{R}$ è una delle relazioni in $\{<, \leq, =, \geq, >\}$;
- v è un valore;
- le variabili assumono valori in $\{0, 1\}$. □

18.3.1 Istanza di problema PLI di riferimento

PLI2QUBO-PLI-piccolo.mp4 dopo aver ricordato la struttura di un problema in PLI, data qui sopra, illustra la sintesi di un modello QUBO sfruttando una semplice istanza di problema in PLI. Lo scopo è illustrare la necessità di trasformare i vincoli disequazionali in vincoli equazionali per mezzo di variabili *slack*, da cui segue la necessità di fissare un intervallo, il più ristretto possibile, dei valori assumibili da ciascuna varaibile *slack*. L'ovvio motivo è contenere le dimensioni dello spazio in cui cercare il soddisfacimento dei vincoli:

- `PLI2QUBO-PLI-piccolo.py` è il sorgente che permette di campionare lo spazio degli stati costituito da 2^7 tuple e trova la risposta al problema QUBO ottenuto grazie alla riduzione illustrata.
- La risposta con energia -17 coincide con quella fornita dall'interpretare con MiniZinc il sorgente `PLI2QUBO-PLI-piccolo.mzn` che esprime e permette di risolvere il problema PLI di partenza.

PLI2QUBO-PLI-piccolo.pdf è il documento di riferimento.

18.3.1.1 Esercizio

In analogia con PLI2QUBO-PLI-piccolo.pdf, il documento PLI2QUBO-PLI-Glover.pdf sviluppa l'esempio della sezione "*General 0/1 Programming*" de [GKD19]. Implementare `PLI2QUBO-PLI-Glover.py`, sia `PLI2QUBO-PLI-Glover.mzn` per un *testing* sulla riduzione proposta.

19.1 KP (come PLI) $\leq_P$ QUBO

KP2QUBO-PLI.mp4 illustra la riduzione della istanza $((10, 10, 12, 18), (2, 4, 6, 9), 16)$ di KP visto come problema in PLI:

$$\begin{aligned} &\text{massimizzare } 10 \cdot x_0 + 10 \cdot x_1 + 12 \cdot x_2 + 18 \cdot x_3 \\ &\text{a patto che } 2 \cdot x_0 + 4 \cdot x_1 + 6 \cdot x_2 + 9 \cdot x_3 \leq 16 \\ &\text{con } x_0, x_1, x_2, x_3 \in \{0, 1\} \ , \end{aligned} \tag{19.1}$$

commentando direttamente `KP2QUBO-PLI.py` utile per quanto segue:

- la riduzione che esso implementa usa una sola variabile *slack*, ipotizzando un'occupazione pari a 15 grazie all'algoritmo Greedy-split;
- richiede la taratura del lagrangiano: rispetto al comportamento dell'`ExactSolver` il lagrangiano deve essere almeno pari a 3. È meno chiaro il valore che esso debba assumere immaginando di usare `SimulatedAnnealingSampler` per trovare la risposta $(x_0 = 1, x_1 = 1, x_2 = 0, x_3 = 1)$ al problema.

19.2 KP $\leq_P$ QUBO ([Luc14])

Questa sezione non fa parte del programma didattico relativo all'A.A.21/22, su cui preparare l'esame.

È per eventuali curiosi che vogliano ricostruire una relazione tra la riduzione KP $\leq_P$QUBO della Sezione 19.1 e la stessa riduzione data in [Luc14].

19.2.1 KP in forma QUBO senza variabili *slack*

KP2QUBO-Lucas-no-slack.mp4, seguendo [Luc14], introduce una riduzione di KP a QUBO, la quale è meno "meccanica" di quella in cui KP è visto come caso particolare PLI.

L'istanza QUBO di riferimento $((p_1, p_2, p_3, p_4), (w_1, w_2, w_3, w_4), W)$ di KP generalizza quella della Sezione 19.1, non specificando i valori di profitti e pesi, ovviamente richiedendo:

$$\begin{aligned} &\text{massimizzare } p_0x_0 + p_1x_1 + p_2x_2 + p_3x_3 \\ &\quad \text{a patto che } w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 \leq W \\ &\qquad \text{con } x_0, x_1, x_2, x_3 \in \{0, 1\} \ . \end{aligned} \tag{19.2}$$

La riduzione in [Luc14] è interessante perché genera il seguente polinomio quadratico multivariato:

$$\begin{aligned} \text{minimizzare } \ &-(p_1x_1 + p_2x_2 + p_3x_3 + p_4x_4) \\ &+ A \cdot (1 - (y_1 + \ldots + y_W))^2 \\ &+ A \cdot ((y_1 + 2y_2 + \ldots + Wy_W) \\ &\qquad - (w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4))^2 \end{aligned} \tag{19.3}$$

su cui valgono le seguenti osservazioni:

- a grandi linee, la progettazione di (19.3) ricorda la strategia che si segue per risolvere il KP per mezzo della *Programmazione Dinamica*, risolvendo l'*all-capacities* KP:
  - le variabili y_i hanno il seguente significato:
    - $y_i = 1$ se si ipotizza che lo zaino potrà essere riempito per una capacità esattamente pari a i;
    - $y_i = 0$ altrimenti.
  - la combinazione dei valori dei seguenti polinomi:

    $$1 - (y_1 + \ldots + y_W) \tag{19.4}$$
    $$(y_1 + 2y_2 + \ldots + Wy_W) - (w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4) \tag{19.5}$$

    che troviamo in (19.3), rappresenta tutti i vincoli che, uno dopo l'altro impongono la possibilità di riempire lo zaino esattamente con un singolo valore w, tale che $0 \leq w \leq W$.
- Il problema (19.3) può fornire una risposata per A sufficientemente grande, da [Luc14] indicato come:

$$A \geq 1 + \max\{p_1, \ldots, p_n\} \ .$$

KPQUBO-Lucas-no-slack.pdf è il documento di riferimento.

19.2.2 KP in forma QUBO con variabili *slack*

- KPQUBO-Lucas-slack.mp4 descrive come ridurre il numero di variabili di una istanza di KP trasformata nella sua versione QUBO in accordo con [Luc14]. Evitare la proliferazione delle variabili in un problema QUBO semplifica il

processo di *minor-embedding* e riduce la possibilità di dover letteralmente spezzare la ricerca delle risposte in due parti che, alla fine, vanno ricomposte.

KPQUBO-Lucas-con-slack.pdf è il documento di riferimento.

`KPQUBO-Lucas-con-slack.py` è il sorgente che implementerebbe(!!) la riduzione con variabili *slack* secondo [Luc14], relativamente all'istanza $((10, 10, 12, 18), (2, 4, 6, 9), 16)$ di KP.

- KPQUBO-LucasVsPLI.mp4 illustra che la riduzione sviluppata nella Sezione 19.1, e quella descritta al punto precedente, coincidono.

  KPQUBO-LucasVsPLI.pdf è il documento di riferimento.

19.3 Un'ultima visione d'insieme su QUBO

"Quantum Bridge Analytics I: A Tutorial on Formulating and Using QUBO Models" [GKD19] fornisce ulteriori esempi di riduzione da problemi classici a problemi QUBO, inquadrando l'argomento in un ambito più ampio, che gli autori identificano come *Quantum Bridge Analytics*, e la cui descrizione è completata da "Quantum Bridge Analytics II: Network Optimization and Combinatorial Chaining for Asset Exchange", lavoro maggiormente orientato alla descrizione dell'uso che si può fare di sistemi ibridi, in cui parte della computazione è classica, e parte quantistica.

Bibliografia

[AL18] T. Albash e D. Lidar. "Adiabatic quantum computation". In: *Reviews of Modern Physics* 90 (2018), p. 015002.

[GJ79] M. R. Garey e D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[GKD19] F. Glover, G. Kochenberger e Y. Du. "Quantum Bridge Analytics I: a tutorial on formulating and using QUBO models". In: *4OR* 17.4 (2019). (https://drive.google.com/file/d/1ofk4trfswCtdbHXTOuloqJfdawMEp8-i/view?usp=sharing), pp. 335–371.

[HSR07] E. Horowitz, S. Sahni e S. Rajasekaran. *Computer Algorithms*. 2nd. Disponibile per il *download* su https://www.pdfdrive.com. Summit, NJ, USA: Silicon Press, 2007.

[Kar72] R. M. Karp. "Reducibility Among Combinatorial Problems." In: *Complexity of Computer Computations*. A cura di R. E. Miller e J. W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103.

[Kar10] R. M. Karp. "Reducibility Among Combinatorial Problems". In: *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. A cura di M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi e L. A. Wolsey. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 219–241.

[KPP04] H. Kellerer, U. Pferschy e D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[Koz] D. Kozen. *The Design and Analisys of Algorithms*. URL: https://www.cs.cornell.edu/~kozen/Papers/daa.pdf.

[Luc14] A. Lucas. "Ising formulations of many NP problems". In: *Frontiers in Physics* 2 (2014). (https://arxiv.org/abs/1302.5843), p. 5.

[McG20] C. McGeoch. "Theory versus practice in annealing-based quantum computing". In: *Theor. Comput. Sci.* 816 (2020), pp. 169–183.

[Tal09] N. N. Taleb. *Il cigno nero. Come l'improbabile governa la nostra vita*. Saggi. Tascabili. Il Saggiatore, 2009.

[Wik] Wikipedia. *List of NP-complete problems*. URL: https://en.wikipedia.org/wiki/List_of_NP-complete_problems (visitato il 25/05/2020).

Parte V

Appendici

Video e testo inclusi per completezza, ma non necessariamente curati al livello necessario.

A.1 Implementazioni *Least-cost* per Subsetsum

- S3Rand-e-S3LC.mp4 inizia col descrivere il metodo `S3Rand.risposte` del *package* `subsetsumBB` che ha la struttura nota adatta a visitare uno spazio degli stati organizzato a sottoinsiemi.

  Ne caratterizzano il comportamento i metodi:

  - `S3Rand.indiceENode` richiamato dal metodo `S3Rand.eNode`. `S3Rand.indiceENode` sceglie l'*live node* che deve essere etichettato *E-node* casualmente, evitando in prima istanza di aderire ad una qualsiasi strategia di visita;
  - `S3Rand.fCompostoH` che calcola il valore $f(h(x[0 \dots j]))$;
  - `S3Rand.stimaCostoPerDifetto` che calcola il valore $\sum_{j \le k < split} X_k$ in cui k *non assume* il valore *split*;
  - `S3Rand.stimaCostoPerEccesso` che calcola il valore $\sum_{j \le k \le split} X_k$ in cui k *assume* il valore *split*.

  Quindi introduce `S3LC.indiceENode` che ridefinisce `S3Rand.indiceENode` in modo da scegliere come *E-node* il *live node* con minimo valore della *funzione costo*.

  Al *minuto 16:30* la presentazione si conclude commentando l'analogia tra quanto implementato dalle classi `S3Rand` e `S3LC` e lo pseudo-algoritmo **8.1** in [HSR07] richiamato in LC-confronto-Horowitz.pdf.

- LC-BranchBound-per-Subsetsum.mp4[1] introduce un criterio di *pruning* derivato dalla *funzione costo* "ad intervallo", seguendo i principi riassunti dallo schema LCBB-schema-per-subsetsum.pdf che sono:
  - non si accetta il sottoinsieme $x[0 \dots j)$ perché il valore calcolato come la più piccola approssimazione per eccesso di s è inferiore ad s o perché il valore calcolato come più grande approssimazione per difetto di s è superiore ad s;
  - si accetta se abbiamo prodotto una soluzione, cioè abbiamo terminato di inserire elementi in $x[0 \dots j)$ e:
  $$\hat{c}(x[0 \dots j)) = f(h(x[0 \dots j))) = s \ .$$
  - continua se:
  $$\left(f(h(x[0 \dots j))) + \sum_{j \le k < split} X_k \right) \le s \le (f(h(x[0 \dots j))) + \hat{g}(x[0 \dots j))) \ .$$

- S3LCBB.mp4[2] inizia con l'illustrare la classe `S3LCBB` che estende la classe `S3LC` ridefinendo il solo metodo `rifiuta` che, assumendo il significato ovvio per le variabili `costoPerDifetto` a `costoPerEccesso`, rifiuta proprio quando s non è compreso tra `costoPerDifetto` a `costoPerEccesso`.

02branchandbound.zip è l'archivio con i sorgenti delle classi discusse.

[1] **Errori.** Nell'albero usato per la simulazione a mano, i costi "18" associati ai nodi nella parte destra dell'albeero che si raggiungono con i seguenti cammini `[f,f]`, `[f,f,t]`, `[f,f,t,t]` vanno sostituiti col valore "20".

Imprecisione. *(minuto 14:14)* Si legge che "31" è la migliore approssimazione per difetto del nodo più a destra del secondo livello , mentre il valore corretto è "20", come scritto.

[2] **Errore.** *(minuto 03:26)* La corrispondenza tra condizione d'uscita del metodo `S3LCBB.rifiuta` e condizioni di rifiuto schematiche sono invertite.