Risposte AlgComp

Lorenzo Dentis, lorenzo.dentis@edu.unito.it

23 maggio 2022

Indice

1 Brute-force e certificazione **2**
1.1 A.1 . . . 2
1.2 A.2 . . . 4
1.3 A.3 . . . 5
1.4 A.4 . . . 6

2 Backtrack **7**
2.1 B.1 . . . 7
2.2 B.2 . . . 8
2.3 B.3 . . . 10

3 Branch&Bound **11**
3.1 C.1 . . . 11
3.2 C.2 . . . 13
3.3 C.3 . . . 16
3.4 C.4 . . . 18
3.5 C.5 . . . 19
3.6 C.6 . . . 20
3.7 C.7 . . . 21
3.8 C.8 . . . 22
3.9 C.9 . . . 24
3.10 C.10 . . . 26
3.11 C.11 . . . 28
3.12 C.12 . . . 30
3.13 C.13 . . . 30
3.14 C.14 . . . 31

4 Programmazione dinamica **34**
4.1 D.1 . . . 34
4.2 D.2 . . . 36

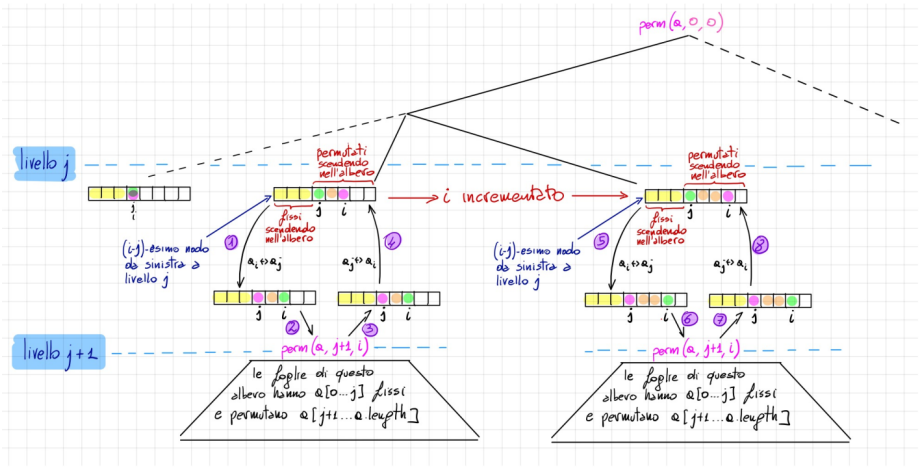
5 Complessità computazionale **38**
5.1 E.1 . . . 38
5.2 E.2 . . . 40
5.3 E.3 . . . 41
5.4 E.4 . . . 42

5.5 E.5 43
5.6 E.6 44
5.7 E.7 45
5.8 E:8 46
5.9 E.9 47
5.10 E.10 48
5.11 E.11 50
5.12 E.12 52
5.13 E.13 53
5.14 E.14 54

6 Problemi computazionali **55**

1 Brute-force e certificazione

1.1 A.1



La figura riassume graficamente la configurazione generica di un algoritmo che genera le permutazioni di una lista di elementi.

Posto un array V di n elementi individuo due indici, i e j.
L'idea è di mantenere gli elementi $[0...j-1]$ fissi ed andare a generare tutte le permutazioni dei restanti elementi $[j...n-1]$grazie all'indice i.

Ad ogni incremento dell'indice i segue una sequenza di chiamate ricorsive che hanno lo scopo di effettuare uno swap fra l'elemto di indice i e l'elemento di indice j e poi incrementare j scendendo nell'albero della ricorsione fino alla condizione $j = i$, quando i viene di nuovo incrementato.Intuitivamente j indica la profondità della ricorsione, i indica l'ampiezza.

- **Punto 1**:*Argomentare a livello intuitivo sul perché la configurazione generica data in figura può essere usata per sintetizzare un algoritmo completo e corretto per la generazione delle permutazioni di una lista di elementi.*

  L'algoritmo è completo in quanto genera $n!$ permutazioni, ed è corretto in quanto sono tutte distinte.

Le permutazioni generate sono esattamente $n!$ in quanto alla "radice" dell'albero avrò n chiamate ricorsive (con $i = 0, i = 1, ...i = n-2, i = n-1$).Invece al livello di profondità j i primi j elementi del vettore risulteranno fissi ed avrò solo $n-1-j$ chiamate ricorsive.
Considerando che il valore di j viene incrementato di una unità ad ogni "livello" di ricorsione fino all' $n-(n-1)$esimo livello ottengo questa semplice equazione:

$$calls = n * (n-1) * ...(j-1\text{volte})... * 1$$

Che non è altro che $n!$
Le permutazioni sono tutte distinte perchè i e j non puntano mai due volte alla stessa cella, posto che ogni elemento del vettore sia distinto.Di conseguenza lo "swap" avverrà sempre tra due elementi differenti ad ogni chiamata ricorsiva, generando sottoalberi uno diverso dall'altro e di conseguenza sequenze differenti.

- **Punto 2**:*Modificare la figura precedente in modo da ottenere la configurazione generica di un algoritmo che genera tutti i sottoinsiemi di elementi di un array, giustificando correttezza e completezza a livello intuitivo.*

  Un algoritmo che generi *tutti i sottoinsiemi di elementi* può essere realizzato organizzando lo spazio degli stati in sottoinsiemi. Al posto che intendere una permutazione come sequenza di elementi la si può vedere come una serie di *scelte*, ovvero booleani.Essenzialmente si va a generare un vettore B di booleani i cui elementi indicano se il corrispondente elemento di V va considerato o meno.
  Se $B[i] == True$ allora $V[i]$ fa parte del sottoinsieme che si sta generando.
  In tal modo otteniamo il **PowerSet** di V, che è completo (cioè le permutazioni sono 2^n) in quanto ogni cella del vettore B può avere solamente valore binario e le celle sono n. E' invece corretto, di nuovo posto che gli elementi siano distinti, in quanto non vi è alcuno *"swap"* e nessuna sequenza è ripetuta più volte. Ogni sequenza di *True/False* varia dalla precedente di una scelta, garantendo sequenze distinte.

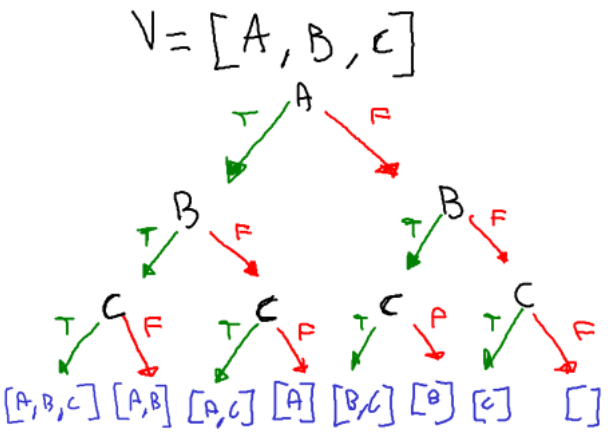


Figura 1: PowerSet di [a,b,c]

1.2 A.2

Fissare una nozione formale di permutazione e dimostrare formalmente che essa è riflessiva e simmetrica.

Definizione formale di una permutazione (definizione ricorsiva):

```
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''
```

Figura 2: Definizione ricorsiva di una permutazione

- perm_nil: [ ] è permutazione di [ ] (lista vuota è permutazione di se stessa)
- perm_swap: $\forall x, y \in Elements$ e $\forall l \in Lists$ $[y :: x :: l]$ e $[x :: y :: l]$ sono una permutazione dell'altra
- perm_skip: dato $x \in Elements$, se $l \in Lists$ è permutazione di $l' \in Lists$ $\Rightarrow [x :: l]$ è permutazione di $[x :: l']$
- perm_trans: date $l, l', l'' \in Lists$ se l permutazione di l' e l' permutazione di $l'' \Rightarrow l$ è permutazione di l''

Teorema (Riflessività)**.** $\forall l \in Lists\ Permutation\ l\ l$

Dimostrazione:

> *caso base*: $l = [\,]$, vale per definizione grazie a perm_nil
> *caso induttivo*: $l = [h :: t]$ con $h \in Lists$ e $t \in ELements$.
> Grazie all'ipotesi induttiva si può affermare $Permutation\ t\ t$ e quindi per perm_skip $Permutation\ [h :: t]\ [h :: t]$. Dato che $l = [h :: t]$ abbiamo che $Permutation\ l\ l$.

Teorema (Simmetria)**.** $\forall l, l' \in Lists,\ Permutation\ l\ l' \Rightarrow Permutation\ l'\ l$

Dimostrazione:

> Assumo che l sia permutazione di l', ma allora devo poter applicare la definizione di *Permutazione*, i 4 casi di figura 2
>
> *primo caso*: $l = []\ l' = []$, la proprietà *perm_nil* ci garantisce sia che $P\ l\ l'$ che $P\ l'\ l$. l' ed l si sono "scambiate"
>
> *secondo caso*: $l = x :: y :: t$, $l' = y :: x :: t$. Assunta vera $P\ l\,l'$, cioè $P\ x :: y :: t\ y :: x :: t$, grazie a *perm_swap* si può affermare $P\ y :: x :: t\ x :: y :: t$, cioè $P\ l'\ l$
>
> *terzo caso*: caso induttivo, è stata applicata *perm_skip* all'ipotesi quindi $l = x :: t$ e $l' = x :: t'$ e per definizione $P\ t\,t'$. Ma per ipotesi induttiva $P\ t\,t' \Rightarrow P\ t'\,t'$, riapplicando *perm_skip* ottengo $P\ x :: t'\ x :: t$ che è esattamente $P\ l'\ l$
>
> *quarto caso*: caso induttivo, è stata applicata *perm_trans* all'ipotesi e quindi deve esistere una lista intermedia l'' t.c. $P\ l\,l''$ e $P\ l''\,l'$ sono

vere. Applicando l' ipotesi induttiva alle due affermazioni precedenti otteniamo:$P\ l''\ l$ e $P\ l'\ l''$.
Applicando *perm_trans* ad entrambe si ottiene $P\ l'\ l$

1.3 A.3

Dimostrare formalmente la completezza di un algoritmo di generazione di permutazioni di una lista di elementi.

Dimostrare che è algoritmo che genera permutazioni di una lista di elementi è completo significa dimostrare che genera $n!$ permutazioni, con n pari al numero di elementi. Formalmente:

Teorema (Completezza)**.** $\forall l \in Lists, len(permutation\ l) = len(l)!$

Dimostrazione:

caso base: $l = [\,]$
In questo caso $len(l)! = 0! = 1$.
Allo stesso modo $len(permutation)\ [\,] = len([\ [\,]\ ]) = 1$.
Quindi $len(permutation\ l) = len(l)!$

Caso induttivo: riprendiamo l'ipotesi induttiva: $len(permutation\ t) = len(t)!$ $l = h :: t$, l'enunciato da dimostrare è

$$len(permutation\ [h :: t]) = len([h :: t])!$$

Risulta ovvio che $len(h :: t) = 1 + len(t)$. Per ipotesi induttiva il numero di liste generate da $permutation\ t$ sarà $len(t)!$.
Per definizione **permutation h::t** calcola tutte le permutazioni di t e poi applica ad ogni lista $l' \in permutation\ t$ l'operazione distribute incrementandole di un elemento.

$$len(h :: t) = len(concat\_map\ (distribute\ h)(permutation\ t))$$

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

Figura 3: Lemma distribute lenght

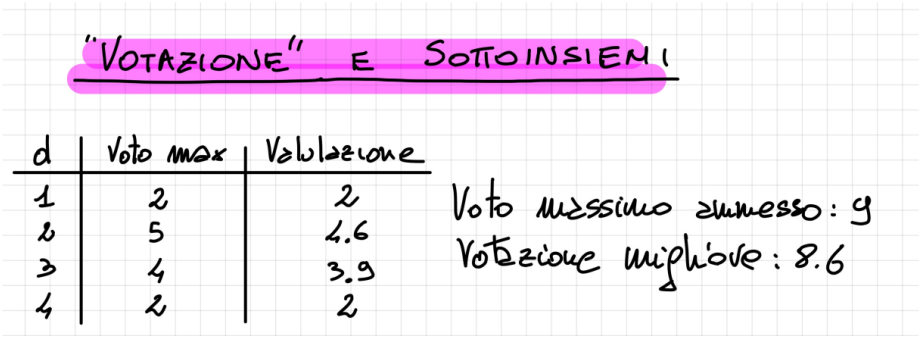
Intuitivamente stiamo ripetendo l'operazione $distribute\ h\ l'$ per $len(t)!$ volte ogni volta producendo una lista di $1 + len(l')$ elementi, quindi otterremo una lista di liste con $(1 + len(l')) * len(t)!$ elementi.
Essendo le liste l' permutazioni di t dovrà valere la sequente equivalenza $len(l') = len(t)$, dato che sto solo eseguendo scambi, non sto aggiungendo ne togliendo elementi.
Sostituendo $len(t)$ a $len(l')$ nell'equazione che specificava il numero di elementi (TODO: fare un ref all'equazione) delle liste otteniamo che $permutation\ h :: t$ genera una lista di liste di $(1+len(t))*len(t)!$ elementi.

Dato che $(1+len(t))*len(t)! = len([h :: t])!$ (poichè un fattoriale (1 +n)* n! = (1 + n)! per definizione) possiamo affermare che:

$$len(permutation\ [h :: t]) = len([h :: t])!$$

Notiamo due cose: Ho affermato che intuitivamente *distribute* h l' ripetuto $len(t)$ volte produce una lista di $1 + len(l')$ elementi, questo andrebbe dimostrato. L'altra affermazione che andrebbe dimostrata è "essendo le liste l' permutazioni di t".Infatti tale affermazione si basa sull'assunto che **permutation t** produca permutazioni di t, che non è nient'altro che il teorema di correttezza delle permutazioni. Quindi la completezza si basa sulla correttezza

1.4 A.4

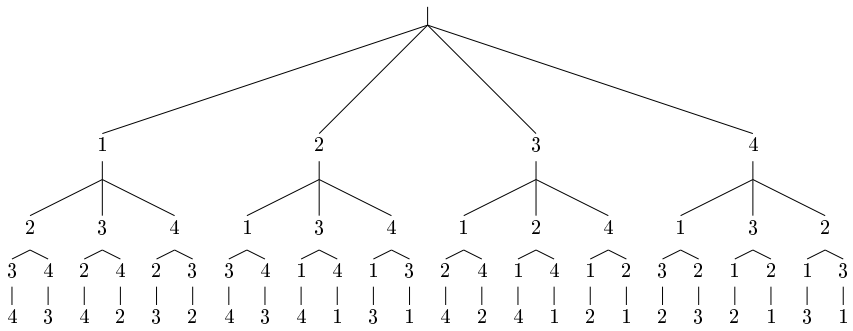


La figura riassume un'istanza del problema Valutazione.

Una *visita esaustiva* anche detta ***Brute-force*** è una strategia di visite che consiste nel verificare tutte le soluzioni teoricamente possibili fino a che si trova quella effettivamente corretta.

- **Permutazioni**:*Descrivere una ricerca Brute-Force di una risposta in uno spazio degli stati organizzato come permutazioni.*

  Si può individuare una *risposta* al problema **Valutazione** generando l'albero di tutte le possibili permutazioni:

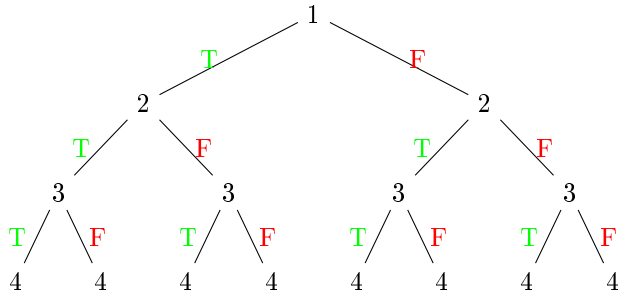


  In questo albero un ramo individua una permutazione dell'insieme di partenza, di conseguenza basta percorrerlo calcolando di volta in volta il voto massimo finchè questi non supera il valore fissato (in questa istanza 9).Si ottiene quindi una soluzione, data dalla sequenza di "voti" ottenuta partendo dalla radice e percorrendo il ramo corrispondente fino al padre del

nodo che causa il superamento del voto massimo.
Attenzione, vengono calcolate tutte le permutazioni, durante la fase di valutazione si va a calcolare il voto e ci si ferma quando il valore viene superato. Se facessimo questo distinguo in fase di generazione delle permutazioni staremmo già facendo pruning.
Eseguendo questa operazione su tutti i rami dell'albero si ottengono tutte le *soluzioni*, scegliendo le soluzioni con votazione migliore si ottengono una o più *risposte*.

- **Sottoinsiemi**:*Descrivere una ricerca Brute-Force della risposta in uno spazio degli stati organizzato a sottoinsiemi.*

  Si può individuare una *risposta* al problema **Valutazione** generando l'albero di tutte le possibili scelte:



  A seconda che si scelga un ramo o l'altro la domanda sarà inclusa nella possibile *soluzione* quindi percorrendo un ramo fino alla foglia si giunge sempre ad una sequenza di voti. Non tutti i sottoinsiemi così generati sono però *soluzioni*, in quanto alcuni superano il vincolo del voto massimo ammesso (in questo caso 9); considerando solo le *soluzioni* e scegliendo quelle aventi la migliore votazioni si ottengono una o più *risposte*

2 Backtrack

2.1 B.1

Illustrare uno pseudo-algoritmo a piacere che implementa la tecnica Backtrack, giustificando le varie parti.

Il **Backtracking** è una tecnica algoritmica atta a migliorare la visita **Brute-force** tramite l'elimnazione di alcuni sotto-alberi dell' albero delle scelte che non vale la pena di analizzare. Più precisamente vengono introdotti due concetti:

- *funzione bound:* una funzione che dato un nodo dell'albero delle scelte fornisce un criterio di valutazione sulla possibilità che il sotto-albero avente come radice quel nodo possa o meno fornire una *risposta*
- *pruning*: Rimozione di sotto-alberi dall'analisi dell'albero delle scelte per il quale la *funzione bound* predice l'impossibilità di generare *risposte*.Perchè violerebbero i vincoli espliciti o perchè produrrebbero una *soluzione* peggiore di altre trovate in precedenza

Il **Backtracking** può essere visto quindi come una visita **BruteForce** con k'aggiunta di un filtro.$BT = BF +$ *funzione criterio/bound* $+$ *pruning* Ogni algoritmo di **Backtracking** sarà quindi così composto:

```
1  BT(a,soluzione)=
2      completa= completa(a,soluzione)
3      if !completa
4          rifiuta=rifiuta(a,soluzione)
5          if !rifiuta
6              soluzioni = espande(a)
7              foreach soluzione in soluzioni
8                  BT(a,soluzione)
9      else
10         accetta = accetta(a,soluzione)
11         if accetta
12             risposta = soluzione
```

- ***completa(a,soluzione)*** (riga 2) è una funzione che resituisce *True* se il cammino dalla radice al nodo attuale può costituire una soluzione. (se fossi nella situazione "sottoinsiemi" la completezza sarebbe data dal superamento dei vincoli espliciti all' iterazione successiva, se fossi nella situazione)
- ***rifiuta(a,soluzione)*** (riga 4) è una funzione che resituisce *True* se il nodo genera un sotto-albero che non porterà sicuramente ad una risposta, perchè viola un vincolo esplicito o perchè genererebbe una soluzione peggiore di una soluzione trovata in precedenza
- ***accetta(a,soluzione)*** (riga 10) è una funzione che resituisce *True* se la soluzione passata come parametro è migliore dell' attuale *risposta* (e quindi può prenderne il posto)
- ***espande(a)***(riga 6) è una funzione che espande il nodo, inserendo i suoi figli tra le possibili soluzioni da visitare e da analizzare. La funzione di **Backtracking** sarà quindi chiamata ricorsivamente su tutte queste possibili soluzioni.
- ***BT(a,soluzione)*** (riga 8): lo scopo di effettuare la chiamata ricorsiva sulle soluzioni appena generate è quello di poter effettuare *pruning* su più sottoalberi possibile, individuando più rapidamente i sottoalberi che contengono *risposte*

2.2 B.2

Illustrare la tecnica algoritmica Backtrack usando i seguenti problemi. Per ciascuno argomentare una funzione bound, prestando attenzione alla terminologia usata per descrivere le parti dello spazio degli stati. Inoltre, relativamente al problema dell'ordinamento, argomentare sul perché è possibile evitare soluzioni basate su Backtrack.

- **ordinamento**: una possibile funzione di bound è $a[j-2] > a[j-1]$, presupponendo un vettore $a_0, ..., a_n$ l'algoritmo si basa sul mantenere un indice j t.c. $a_0 \leq ... \leq a_{j-2}$ mentre gli elementi $a_j, ..., a_n$ non sono ancora

stati analizzati. Notiamo però che non ha molto senso utilizzare tecniche **BruteForce** o derivate, perchè l'operazione di confronto fra due numeri può essere sempre effettuata con consto minimo.Potendo confrontare ogni elemento con tutti gli altri elementi posso cancellare una grossa parte dello spazio delle soluzioni. L'esempio più sciocco di questa cosa è il **Boublesort** che , pur confrontando ogni elemento dell'insieme con tutti gli altri, è più efficiente di un approccio **BruteForce**

- **Cammino Hamiltoniano**: formalmente possiamo definire un cammino Hamiltoniano di un grafo G come una sequenza di archi $SE \subset E, G = (V, E)$ che permette di visitare ogni nodo V esattamente una volta. La prima cosa che si può notare è che in questo caso una soluzione equivale ad una risposta, una volta trovato il primo cammino si può effettuare *pruning* di tutto il restante spazio degli stati.
  Una buona *funzione bound* è una funzione che interrompe l'esplorazione quando si cerca di inserire un arco non esistente.partendo da un nodo qualsiasi e scegliendo di visitare un altro nodo possiamo sicuramente escludere tutti i nodi che non sono collegati al nodo di partenza da alcun arco, stessa cosa iterando sul prossimo nodo.Bisogna prestare attenzione al fatto che per come è costruito lo spazio degli stati non è possibile visitare un nodo "già visitato".

- **Colorazione di un grafo**: Lo scopo del problema è assegnare un colore $c \in C$ ad ogni nodo $v \in V$ di un grafo $G = (V, E)$. in modo che vertici adiacenti abbiano colore diverso.
  Supponiamo di aver colorato fino al nodo j e di avergli dato il colore c_a. Guardando il corrispettivo stato nella raffigurazione ad albero dello spazio degli stati (Figura 4) i suoi figli saranno tutti i possibili colori di un altro nodo x.Se scelgo di scendere lungo il ramo contrassegnato dal colore c_a mi devo assicurare che tra il nodo j ed il nodo x non vi sia alcun arco.
  Se scelgo di scendere verso uno stato contrassegnato da un colore diverso da c_a posso non fare questo controllo.
  In entrambi i casi però devo controllare che tutti i nodi connessi da un arco a x abbiano un colore diverso dal colore assegnatogli.

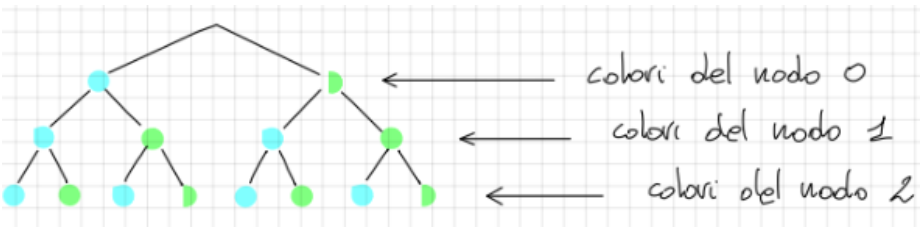


Figura 4: Albero delle scelte per colorazione

- **Subsetsum**: 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. In questo caso la *funzione bound* si basa su tre concetti:

1. se ho trovato una *soluzione* smetto di cercare, sto cercando un generico sottoinsieme, non ho altri vincoli
2. se la somma degli elementi inseriti nella *potenziale soluzione* Z (la soluzione in fase di costruzione) è maggiore di S non proseguo.
3. se i primi due punti non sono veri sommo agli elementi presenti nella mia *potenziale soluzione* Z tutti gli altri elementi dell'insieme X, se il risultato è comunque minore di S non ha senso proseguire.Anche inserendo tutti gli elementi rimasti non otterrei comunque S

2.3 B.3

Le figure riportano proposte in [HSR07] di pseudo-algoritmi di Backtrack. Commentare la funzionalità delle loro parti in base ad una qualche struttura dati fissata per la rappresentazione dello spazio degli stati su cui opera. La sostanziale

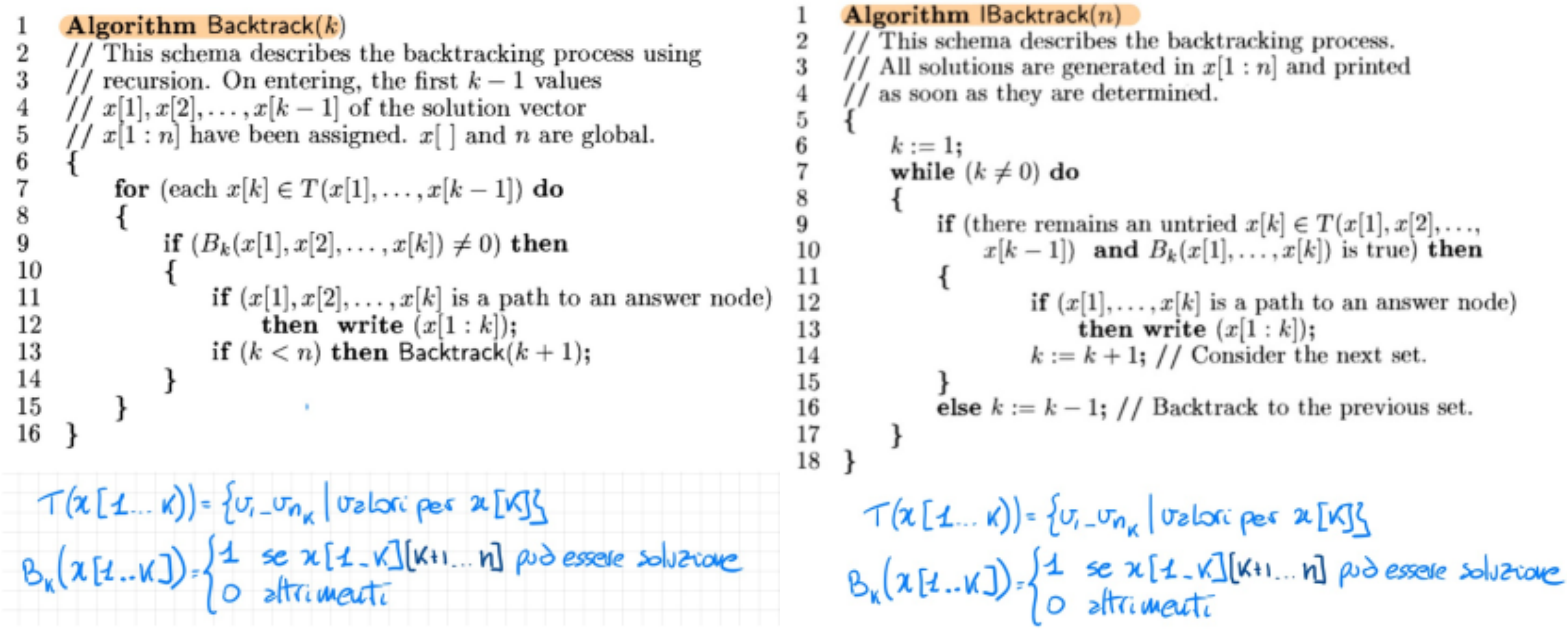


Figura 5: Backtracking ricorsivo ed iterativo

differenza tra i due algoritmi sta in come viene generato lo *spazio degli stati*, in un caso ricorsivamente, nell'altro tramite un algoritmo iterativo.
Questa distinzione è importante in quanto la versione ricorsiva fornisce uno stack implicito, assente nella versione iterativa.Bisogna quindi implementare una struttura dati che permetta di effettuare le operazioni che nel libro (riga 9-10) vengono definite in maniera informale.

Preso $(x_1, x_2, ...x_i)$ path dalla radice ad un nodo dell'albero rappresentante lo spazio degli stati definiamo la *funzine bound* B_k come la funzione che restituisce falso quando il path anche se esteso non raggiungerà mai una *soluzione*.
$T(x_1, ...x_i)$ è il set di tutti i possibili valori di x_{i+1} tali che $(x_1, ...x_{i+1})$ è ancora uno stato da analizzare (non è una soluzione).Quindi $T(x-1, ..., x_n) = \emptyset$

- **Algoritmo ricorsivo**: Il vettore delle soluzioni $(x_1, ..., x_n)$ è gestito come un array globale $x[1:n]$, $x[1:k]$ contiene tutti gli elementi "accettati", corrisponde alla vettore in corso d'opera che potrebbe portarci ad una *soluzione*.
  L'algoritmo prova ad inserire gli elementi rimanenti in k-esima posizione, se il vettore soluzione $(x_1, ..., x_k)$ soddisfa B_k (sappiamo che il vettore è soluzione poichè altrimenti sarebbe stato rifiutato a riga 9) si controlla se tale se è una *risposta temporanea* (riga 11) e la si "stampa", se non lo viene invocata la funzione *Backtrack* ricorsivamente sul nuovo vettore "in costruzione".
  Se sono stati inseriti tutti gli elementi e non si è arrivati ad una soluzione (riga 13 $k = n$) bisogna cambiare l'ultimo elemento (figurativamente è come essere al penultimo livello dell'albero e sceglire una foglia sbagliata).

- **Algoritmo iterativo**:
  Non avendo uno stack implicito bisogna mantenere un vettore T che contenga tutti i possibili valori che possono essere componenti della soluzione.
  All'inizializzazione $T()$ contiene tutti i possibili valori che è possibile posizionare come primo elemento.
  Si tiene conto di quali valori sono stati tentati e quali no, se un valore che non è stato ancora provato soddisfa B_k viene inserito in T (riga 9-10), se il nuovo vettore T è una soluzione viene "stampato" altrimenti si procede nella ricerca come nel caso precedente.
  Si noti che questa non è una visita in ampiezza, si sta comunque scendendo su un solo ramo, quiando tutti i valori che potrebbero stare in x_k sono stati tentati e nessuno di loro ha portato a soluzioni k viene decrementato "tornando indietro" e modificando il valore dell'elemento precedente.

3 Branch&Bound

3.1 C.1

Ad esempio, anche sfruttando almeno una delle seguenti immagini, definire le nozioni dead node, live node, E-node, descrivere l'impatto delle politiche di generazione di live node e di assegnazione dello stato di E-node sulla visita dello spazio degli stati.

- *dead node*: radice di sotto-alberi che non sono più espandibili, perchè completi o a causa del pruning
- *live node*: nodo "vivo", cioè alcuni dei suoi figli sono ancora da espandere
- *E-node*: L' *expanded node* è il nodo che si sta visitando in questo momento, cioè il nodo del quale si sta per generare un figlio.L' *E-node* e per definizione un *live node*.

A seconda di come si gestisce la visita dei nodi e l'assegnamento dell' *E-node* si può generare una visita dello spazio degli stati differente.

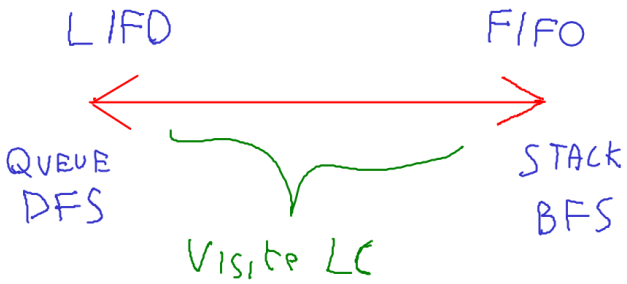


Figura 6: criteri di scelta

Come mostrato in figura 6 FIFO e LIFO sono "i due estremi di un ventaglio di possibili criteri di scelta". Gestendo i live node com uno *stack* e quindi andando ad estrarre l'ultimo nodo inserito si ottiene una visita *depth-first* in cui si trasferisce la condizione di essere *E-node* dal padre al figlio appena generato. In tal modo un nodo non diventa "*dead*" finchè tutti i suoi figli non sono stati visitati, di conseguenza si tende ad arrivare al fondo dell'albero degli stati il più in fretta possibile.

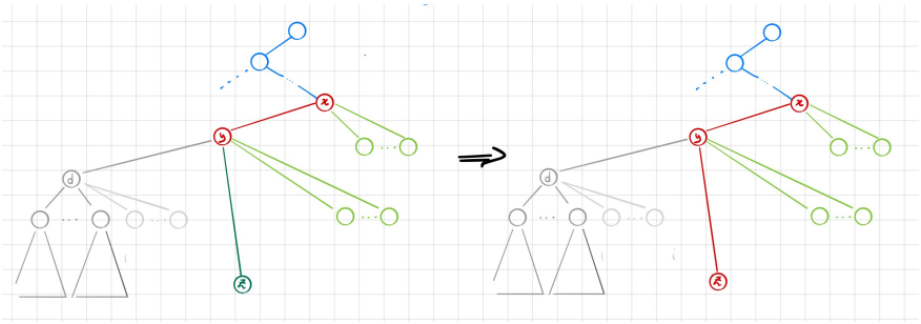


Figura 7: visita in profondità

In maniera diametralmente opposta funziona la gestione dei nodi con una *queue* in cui il primo nodo inserito è il primo ad essere estratto. Questa politica genera una visita *breadth-first* in cui prima di passare ai nodi del livello successivo i nodi del livello attuale vengono tutti visitati, una volta generati tutti i figli di un nodo questi diventa "dead". Ciò porta a visite dello spazio degli stati che prima di giungere alle foglie dell'albero degli stati "esplorano" tutte le possibilità.
Come anticipato in figura 6, tra DFS e BFS si trovano tutte le possibili politiche intermedie, tra cui le *least cost*. Naturalmente l'individuazione di una politica *least cost* è fortemente dipendente dal problema.

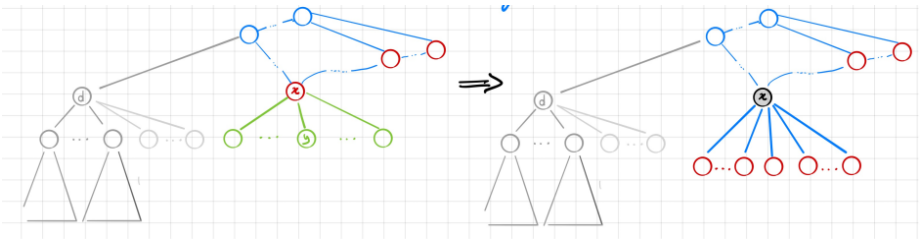


Figura 8: visita in ampiezza

3.2 C.2

L'albero degli stati del problema 4Regine è riportato, che è la Figura 7.2 de [HSR07].

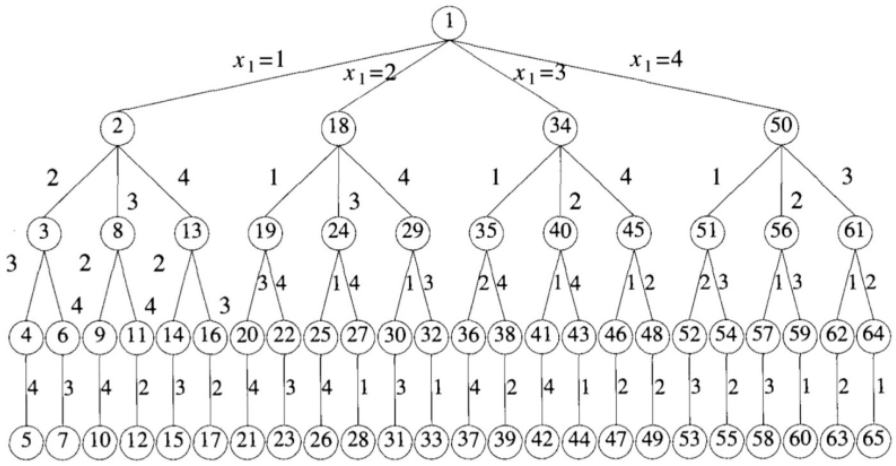


Figura 9: spazio degli stati 4 regine

Punto 1:*Illustrare a grandi linee come si sviluppa la visita dello spazio degli stati qui sopra illustrato, che possiamo definire in profondità ed in ampiezza.*

Presupponiamo di visitare l'albero degli stati tramite le due diverse politiche, effettuando pruning quando necessario.

- **DFS**: La DFS cerca di giungere il prima possibile ad una soluzione, quindi come mostrato in figura 10 si scende lungo il ramo più "a sinistra" finchè non si incontra una violazione dei vincoli che induce il pruning (nel nodo 3 ad esempio abbiamo regine in A4 e B3, sulla stessa diagonale). Il lato negativo è che spesso la visita in profondità si "infila" in cammini che non portano ad una soluzione, come tutto il sottoalbero generato dalla regina in posizione 2 (A4), potrebbe sembrare conveniente visitare più opzioni possibile in modo da eliminare i cammini che non producono soluzioni il

prima possibile.

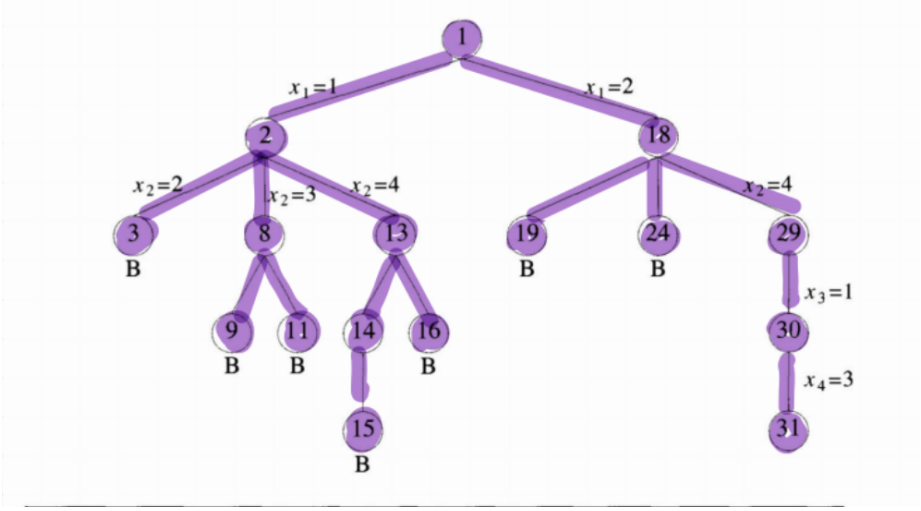


Figura 10: Soluzione con DFS

- **BFS**:Tramite la visita in ampiezza invece si va ad espandere ogni nodo di un livello dell'albero prima di passare al livello successivo. In generale la politica *FIFO* butta via in un sol colpo un sacco di non soluzioni, ma in questo caso prima di incontrare una violazione di qualche vincolo bisogna scendere fino al secondo livello dell'albero, analizzando 16 nodi (che è esattamente il numero di nodi necessari a trovare la **risposta** con la visita in profondità). La politica *LIFO* si dimostra quindi molto meno vantaggiosa, oltretutto si porta dietro un altro svantaggio: se infatti la *FIFO* approfittava dello stack generato dalla ricorsione per mantenere i suoi progressi in memoria nella *LIFO* bisogna costruire tante strutture dati quanti sono i live node atte a mantenere i nodi della "frontiera" in memoria, dato che ogni soluzione viene costruita in parallelo.

Punto 2:*fornire un riscontro del fatto che visite in ampiezza non sono necessariamente più efficaci di quelle in profondità per individuare una risposta ed argomentare sulla inapplicabilità di criteri "ovvi" di ranking dei live node, per realizzare strategie di visita di uno spazio degli stati, né in ampiezza, né in profondità..*

Confrontando le due immagini presenti in figura 12 notiamo chiaramente che la visita *DFS* è più vantaggiosa in questo caso, giungendo alla soluzione in 16 passi rispetto ai 29 della visita *BFS*.

In generale è vero che la politica *FIFO* permette di "buttare via" in un sol colpo un sacco di non soluzioni, ma in questo caso prima di giungere a nodi non accettabili si è obbligati a visitare gran parte dell'albero degli stati. Sto sì eliminando moltissimi sotto-alberi rispetto alla *DFS* ma per

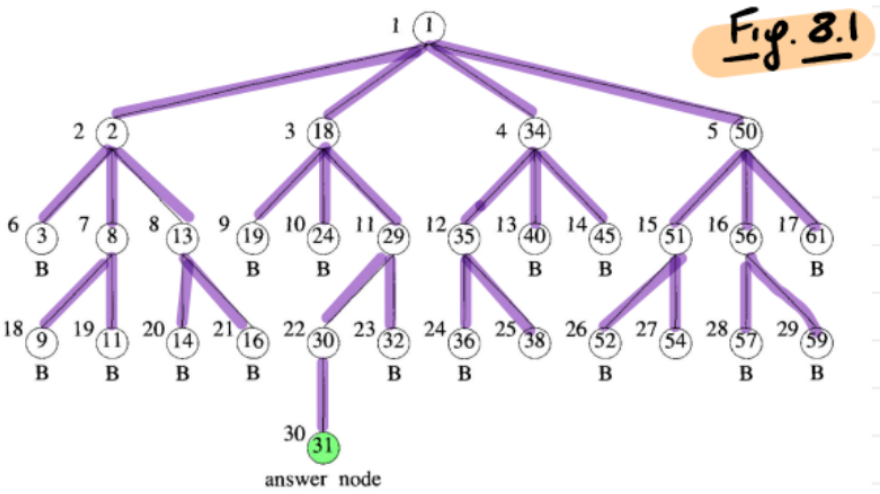


Figura 11: Soluzione con BFS

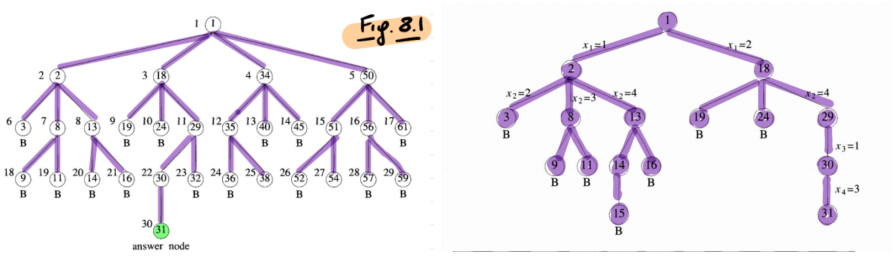


Figura 12: confronto visita in profondità ed ampiezza

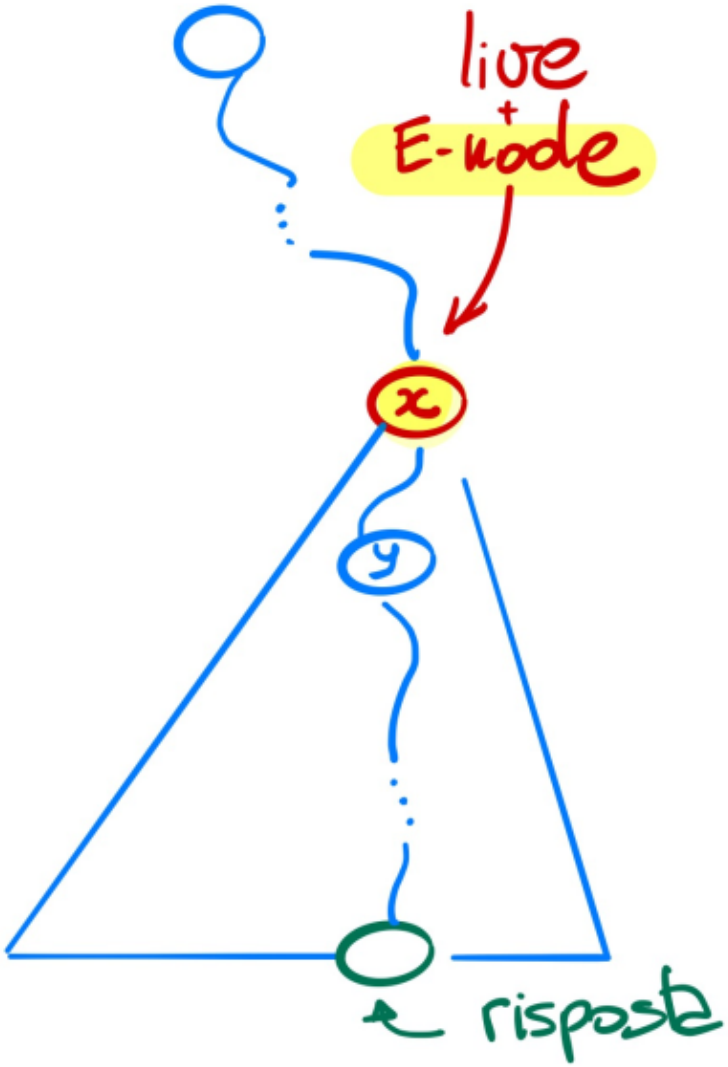
farlo ne sto esplorando molti di più.Più l'albero è ampio rispetto alla sua profondità peggiori sono le prestazioni di una visita in *ampiezza*

L'ideale sarebbe visitare in ampiezza ma scendere in profondità quando un ramo sembra promettente, per individuare i rami più promettenti si potrebbe ad esempio ordinare i live node minimizzando una delle seguenti proprietà:

1. Il numero di nodi nel sottoalbero di x da generare prima di ottenere una *soluzione*.
2. il numero di livelli che separano il live node dalla risposta.

Il problema derivante dal tentare questo approccio sta proprio nel calcolare le due misure. Non conoscendo le *soluzioni*, e di conseguenza nemmeno la *risposta*, l'unico modo di effettuare le misure descritte sopra sarebbe quello di esplorare lo spazio degli stati fino ad una *risposta*. Questa situazione è però paradossale, se conoscessi la risposta non la starei cercando, quindi la misurazione delle due proprietà non è un criterio di ranking applicabile.

3.3 C.3



Considerare, ad esempio, il seguente schema riassuntivo

Punto 1 *Argomentare sul significato di una funzione costo per individuare una risposta, e ricostruire la struttura di una funzione costo approssimata.*

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

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

- h è una funzione che quantifica il lavoro, quindi $h(x)$ è il lavoro noto svolto fin'ora per arrivare al nodo x (quindi proporzionale alla distanza dalla radice al nodo)

- f è la funzione *peso* monotona, dato in input il lavoro ottenuto da h fornisce un valore proporzionale ad esso utilizzabile come *peso*
- $\hat{g}$ è una *stima* della funzione non nota g.Essendo impraticabile calcolare il valore effettivo $g(x)$, $\hat{g}(x)$ fornisce una stima di quello che sarà il costo della costruzione di un cammino dal nodo x fino ad una eventuale *risposta* che il sotto-albero di radice x contiene.
  la funzione $\hat{g}$ non è generalizzabile, dipende dal problema.

E' necessario introdurre una funzione costo in quanto non è possible calcolare accuratamente il *ranking* di un nodo, bisogna quindi ripiegare su una stima che permetta scegliere in maniera intelligente il prossimo *E-node*. E' composta principalmente da due parti, il *costo noto* ed il *costo stimato*

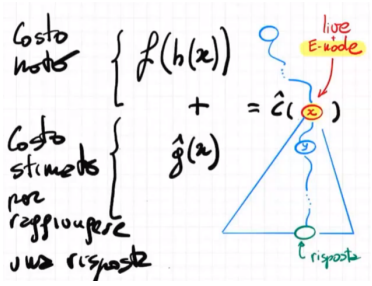


Figura 13: funzione costo

Notiamo anche che dati due nodi x, y se y è un discendente di x ci aspettiamo che la funzione $\hat{g}(x) > \hat{g}(y)$, altrimenti la funzione costo è mal formata.

punto 2 *Dopo aver ricordato la struttura di una funzione costo approssimata che quantifichi sia il costo del lavoro di ricerca della risposta già effettuato, e che stimi quello ancora da fare, argomentare le conseguenze di considerare nullo il costo del lavoro già svolto.*

La *funzione costo* $\hat{c}(x)$ di un *live node* x è una funzione atta a individuare il nodo più conveniente da far diventare *E-node* $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ 3.3 Analizziamo la situazione: $f(h(x)) = 0$, in questo caso il *costo noto* non ha influenza sul risultato generato da $\hat{c}(x)$ e la *funzione costo* può essere riscritta come

$$\hat{c}(x) = \hat{g}(x)$$

Questo ci riporta ad un algoritmo *DFS*, in quanto non vi è "penalità" nell' espandere nodi in profondità nell'albero degli stati.Considerando che dati due nodi x, y se y è un discendente di x allora $\hat{g}(x) > \hat{g}(y)$ la scelta che minimizza la *funzione costo* è sempre quella che espande un nodo di livello successivo, quindi una *D-search*. Immaginiamo ad esempio due *live node* x, z, l' *E-node* è x e genera due discendenti: w_1, w_2.

$0 + \hat{g}(x) < 0 + \hat{g}(z)$ dato che x è *E-node*
$0 + \hat{g}(w_1) \leq 0 + \hat{g}(w_2) < 0 + \hat{g}(z)$ dato che $\hat{g}$ è monotona
$\hat{c}(w_1) \leq \hat{c}(w_2) < \hat{c}(z)$

Di conseguenza il prossimo *E-node* sarà scelto tra w_1 e w_2

punto 3 *Dopo aver ricordato la struttura di una funzione costo approssimata che quantifichi sia il costo del lavoro di ricerca della risposta già effettuato, e che stimi quello ancora da fare, argomentare le conseguenze di considerare non nullo il costo del lavoro già svolto.*

La *funzione costo* $\hat{c}(x)$ di un *live node* x è una funzione atta a individuare il nodo più conveniente da far diventare *E-node* $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ 3.3
Analizziamo la situazione: $f(h(x)) \neq 0$, $\hat{c}(x) = f(h(x)) + \hat{g}(x)$.
Riprendendo l'idea del punto precendente abbiamo due *live node* x, z, l'*E-node* è x e genera discendenti chiamati $w_1, ..., w_n$.

$$f(h(x)) + \hat{g}(x) < f(h(x)) + \hat{g}(z)$$
$$c(w_1) = (f(h(x)+\Delta) + (\hat{g}(x) - \Delta')$$
$$c(w_1) = (f(h(x))+\hat{g}(x)) + (\Delta - \Delta')$$

$$\begin{cases} se\ (\Delta - \Delta') & \leq 0 \text{ proseguiamo lungo il ramo di } w_1 \\ se\ (\Delta >> \Delta') & w_1 \text{ non è un nodo conveniente} \end{cases}$$

In questa situazione i valori di $\hat{c}(w_1), \hat{c}(w_2), ..., \hat{c}(w_n)$ possono superare quelli relativi a live node inseriti nella lista prima di $w_1, w_2, ..., w_n$ e, quindi, risultare sconvenienti da seguire rispetto ad altri.

3.4 C.4

Inquadrare e descrivere la relazione tra le componenti della funzione costo $\hat{c}(x)$ *e le strategie Breadth-search e D-search in uno spazio degli stati.*

Come già mostrato in sezione 3.1, più precisamente in figura 6 *FIFO* e *LIFO* sono "i due estremi di un ventaglio di possibili criteri di scelta" ed in quanto tali possono essere ottenuti tramite una precisa formulazione della *funzione costo*. Infatti analizzando la *funzione costo* $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ si nota facilmente quali sono le possibili casistiche che possono portare ad un estremo o all'altro.

- $\hat{c}(x) = f(h(x)) + 0$ In questo caso la *funzione costo* non subisce l'influenza del *costo stimato*. La *funzione costo* può essere riscritta come

  $$\hat{c}(x) = f(h(x))$$

  Intuitivamente i *nodi* più vicini alla radice avranno un costo inferiore, di conseguenza verranno scelti per primi e si verificherà la classica situazione caratteristica delle *Breadth-search*: tutti i nodi appartenenti ad un livello dell'albero verranno esaminati prima di procedere con l' espansione dei nodi del livello successivo. Immaginiamo ad esempio due *live node* x, z, l' *E-node* è x e genera due discendenti: w_1, w_2.

  $$\begin{aligned} &f(h(x)) + 0 < f(h(z)) + 0 \text{ dato che } x \text{ è } \textit{E-node} \\ &f(h(w_1)) + 0 = f(h(w_1)) + 0 > f(h(z)) + 0 \qquad (1) \\ &\hat{c}(w_1) = c(w_2) > \hat{c}(z) \end{aligned}$$

  Di conseguenza il prossimo *E-node* sarà sicuramente z.
  L'equazione del punto (1) deriva dal fatto che per giungere a w_1, w_2 è stato fatto del lavoro in più, esattamente il lavoro necessario ad espandere x

- $\hat{c}(x) = 0 + \hat{g}(x)$ Questa casistica, già affrontata nella sezione 3.3 al **Punto 2** si riconduce ad una *D-search*. Se $f(h(x)) = 0$ il *costo noto* non ha influenza sul risultato generato da $\hat{c}(x)$ e la *funzione costo* può essere riscritta come:

  $$\hat{c}(x) = \hat{g}(x)$$

  Questo ci riporta ad un algoritmo di *DFS*, in quanto non vi è "penalità" nell' espandere nodi in profondità nell'albero degli stati. Considerando che dati due nodi x, y se y è un discendente di x allora $\hat{g}(x) > \hat{g}(y)$ la scelta che minimizza la *funzione costo* è sempre quella che espande un nodo di livello successivo, quindi una *D-search*. Immaginiamo ad esempio due *live node* x, z, l' *E-node* è x e genera due discendenti: w_1, w_2.

  $$0 + \hat{g}(x) < 0 + \hat{g}(z) \text{ dato che } x \text{ è } \textit{E-node}$$
  $$0 + \hat{g}(w_1) \leq 0 + \hat{g}(w_2) < 0 + \hat{g}(z) \text{ poichè } \hat{g} \text{ è monotona}$$
  $$\hat{c}(w_1) \leq \hat{c}(w_2) < \hat{c}(z)$$

  Di conseguenza il prossimo *E-node* sarà scelto tra w_1 e w_2

3.5 C.5

La seguente immagine riporta l'Algoritmo 8.1 in [HSR07]. Commentarne il funzionamento in base ad una qualche struttura dati fissata per la rappresentazione dello spazio degli stati su cui opera.

L'algoritmo in figura 14 rappresenta un generico algoritmo per una visita *Least Cost*.

La struttura dati su cui si appoggia è una lista di *record*, in cui ogni record può essere visto come un "nodo" dell'albero che rappresenta lo spazio degli stati.
La struttura dati *listnode* fornisce due operazioni `Add(x)` e `Least()` che forniscono la possibilità di inserire o estrarre record. Possiamo assumere che l'operazione `Least()` estrae dalla struttura dati il nodo che ha il minimo costo tra i live node $\hat{c}(\cdot)$. A seconda di come sono è implementata l'operazione `Add(x)` avremo differenti comportamenti nel caso si incontrassero nodi aventi lo stesso *costo*, allo stesso modo di quanto discuso in sezione 3.3 e sezione 3.4.

Analisi dell'algoritmo:

- Le righe 4-6 inizializzano la struttura dati ed impostano la radice dell'albero come ***E-node***.
- In riga 7 viene iniziato un ciclo che si concluderà quando tutto lo spazio degli stati sarà stato completamente esplorato oppure sarà stata individuata una *risposta*, non vi è alcun criterio di *pruning*
- Le righe 9-15 sono il cuore della ricerca, vengono generati tutti i figli dell' ***E-node***, se nessuno di loro è *risposta* vengono aggiunti alla lista dei *live node* tramite l'operazione `Add(x)`.
- Quando tutti i figli di un nodo sono stati aggiunti alla *listnode* il nodo è sostanzialmente diventato un *Dead node* ed il nodo che fornisce il minimo valore della *funzione costo* viene scelto come prossimo ***E-node***.

```
listnode = record {
    listnode *next, *parent; float cost;
}

1   Algorithm LCSearch(t)
2   // Search t for an answer node.
3   {
4       if *t is an answer node then output *t and return;
5       E := t; // E-node.
6       Initialize the list of live nodes to be empty;
7       repeat
8       {
9           for each child x of E do
10          {
11              if x is an answer node then output the path
12                  from x to t and return;
13              Add(x); // x is a new live node.
14              (x → parent) := E; // Pointer for path to root
15          }
16          if there are no more live nodes then
17          {
18              write ("No answer node"); return;
19          }
20          E := Least();
21      } until (false);
22  }
```

Figura 14: Algoritmo Least Cost

3.6 C.6

Definire una funzione costo per il problema ***Subsetsum*** *e discuterne le possibili interazioni con meccanismi* ***FIFO/LIFO*** *di accumulo dei live node e con criteri di pruning.*

L'idea da cui partire per sviluppare una *funzione costo* per ***SubsetSum*** è quella di individuare un intervallo di valori al cui interno è compreso il valore s del problema ***SubsetSum*** $X = (X_0 ... X_{n-1}, S)$ Per chiarezza presentiamo con X_k il valore numerico di un elemento, $x_k \in 0, 1$ indica se il numero è inserito nell'insieme o no. Come descritto nella sezione 3.3 la *funzione costo* $\hat{c}$ è composta da una *parte nota* ed una *parte stimata*.

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

- La parte nota è facilmente individuabile come la somma dei numeri già inseriti nell'insieme.

$$f(h(x[0..j))) = \sum_{0 \leq k < j} X_k x_k$$

- La parte stimata viene calcolata in modo che valga la seguente condizione:

$$f(h(x[0..j))) + \hat{g}(x[0..j)) - V \leq S \leq f(h(x[0..j))) + \hat{g}(x[0..j))$$

  Rimuovendo V ottengo una approssimazione per difetto di S, se non lo rimuovo ottengo una approssimazione per eccesso. Se però V viene preso

dall'insieme X diventa l'elemento che se tolto dalla somma degli elementi che mi rimangono da inserire fornisce la migliore approssimazione per difetto/eccesso di S

$$\hat{g}(x[0..j)) = \sum_{j \leq k < split} X_k$$

in cui il valore split è t.c:

$$\sum_{j \leq k < split-1} X_k \leq S \leq \sum_{j \leq k < split} X_k$$

Lo scopo di questa funzione costo è esplorare in maniera più intelligente lo spazio delle soluzioni, la strategia LC diventa la strategia di scelta del prossimo *E-node* tra tutti i possibili *live-node*. Ma a parità di ***costo*** bisogna comunque affidarsi ad uno dei due criteri (*LIFO/FIFO*).
A seconda di come sono è implementata l'operazione di aggiunta avremo differenti comportamenti nel caso si incontrassero nodi aventi lo stesso *costo*, allo stesso modo di quanto discuso in sezione 3.3 e sezione 3.4. ma si potrebbe pensare anche di non affidarsi ad un criterio fisso e decidere randomicamente di volta in volta, per introdurre "perturbazioni" che riducano il rischio di svolgere ripetutamente scelte sbagliate. E naturalmente anche possibile migliorare l'efficenza di questa visita tramite pruning rimuovendo i rami dell'albero il cui valore $f(h(x[0..j))) + \hat{g}(x[0..j)) - V > S$.Non ha infatti senso esplorare i sottoalberi che generano, in quanto non possono contenere soluzioni.

3.7 C.7

*Definire una funzione costo per il problema **4Regine** e discuterne le possibili interazioni con meccanismi **FIFO/LIFO** di accumulo dei live node e con meccanismi di pruning.*

Per semplicità di comprensione individuiamo la n-esima casella con le sue indicazioni di riga e colonna $n = i * 4 + j$, quindi la vediamo come una matrice.

L'algoritmo quando posiziona una regina su una cella incrementa di 1 il valore di tutte le caselle che la regina può "mangiare" quindi tutte le caselle sulla stessa riga, tutte le caselle sulla stessa colonna e le righe sulle due diagonali.

- La parte nota è data dal numero di caselle della scacchiera occupabili, quindi le celle in nessuna regina è presente o può muoversi e di conseguenza mangiare.

- la stima $\hat{g}$ è data dalla somma del valore di tutte le caselle adiacenti alla casella che si sta analizzano cambiata di segno.

Si può facilmente effettuare pruning, andando a rimuovere i sotto-alberi generati dai nodi che rappresentano l'inserimento di regine su caselle $x \in X$ aventi $x_{ij} \neq 0$.

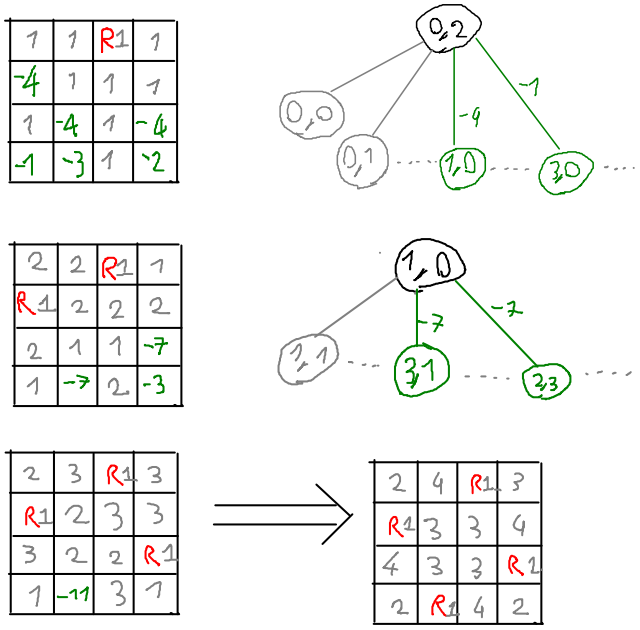


Figura 15: Algoritmo Least Cost 4-regine

Formalmente, identifichiamo la cella sulla i-esima riga e j-esima colonna con x_{ij} ed chiamiamo x^c_{ij} la variabile che indica se la cella è occupata o no.

$$x^c_{ij} \begin{cases} 1 \; se \; x_{ij} = 0 \\ 0 \; se \; x_{ij} > 0 \end{cases}$$

Allora

$$\hat{c}(x_{ij}) = \sum_{i=0}^{n} \sum_{j=0}^{n} x^c_{ij} + \sum_{i=i-1}^{i+1} \sum_{j=j-1}^{i+1} x_{ij}$$

Come in sezione 3.6 a parità di ***costo*** bisogna comunque affidarsi ad uno dei due criteri (*LIFO/FIFO*). Ad esempio al secondo passo è stato scelto di posizionare la regina nella cella x_{01} ma la si sarebbe potuta inserire anche in x_{12} o x_{23}. Notiamo che se la regina fosse stati inserita in x_{12} non avremmo potuto ottenere una soluzione.

3.8 C.8

Illustrare gli algoritmi ***Greedy*** *e* ***Greedy-split****. Discutere la qualità della soluzione offerta da* ***Greedy*** *al variare del valore M, se applicato all'istanza* ***KP***

La qualità di tutti gli algoritmi greedy può essere migliorata ordinando gli elementi in base al rapporto $\frac{valore}{peso}$, operazione con costo $O(n \log_n)$. In generale le approssimazioni fornite da algoritmi greedy per la risoluzione di ***Knapspack*** sono molto vicine alla risposta. Per entrambi gli algoritmi presupponiamo tre

strutture dati di dimensione n, "vettore" w dei pesi (che rappresenta l'isieme dei pesi degli elementi), "vettore" p dei profitti (che rappresenta l'isieme dei pesi degli elementi) e vettore x delle scelte. $x[j] \in 0, 1$ indica se l'elemento di indice j è inserito o no nello zaino.

- **Greedy**:

```
w_tot = 0
z = 0
for j = 0 to n do
        if w_tot + w[j] <= c then
                x[j] = 1
                w_tot = w_tot + w[j]
                z = z + p[j]
        else x[j] = 0
```

  L'algoritmo può essere intuitivamente riassunto in 3 passi

  1. prova ad inserire un elemento di peso $w[j]$ nello zaino
  2. se l' aggiunta dell' elemento non causa il superamento del limite c inseriscilo, passa al prossimo altrimenti.
  3. analizzati tutti gli elementi concludi.

- **Greedy-split**:

```
w_tot = 0
z = 0
j = 0
while w_tot + w[j] <= c
                x[j] = 1
                w_tot = w_tot + w[j]
                z = z + p[j]
                j = j + 1
```

  Questo algoritmo è ancora meno preciso dell' algoritmo **greedy**, in quanto interrompe la ricerca appena si incontra un elemento che causa il superamento del limite c. E' però anche più efficiente, non dovendo analizzare altri elementi che rischiano di non poter essere inseriti nello zaino.

  1. prova ad inserire un elemento di peso $w[j]$ nello zaino
  2. se l' aggiunta dell' elemento non causa il superamento del limite c inseriscilo, altrimenti concludi la visita.

Entrambi gli algoritmi soffrono di un problema, sono *potenzialmente pessimi*, ma non è questo il caso presentato nella domanda.

$$(\vec{w} = (1, M)), \vec{p} = (3, M), 2M$$

Questo caso ha un particolarità, analizzando i pesi si può notare che $p_{max} = M+1$ essendoci solo due elementi, uno di peso M l'altro di peso 1.La particolarità è che $2M > M + 1 \; \forall M > 0$, quindi se lo zaino ha una capienza maggiore di 0 potrà sempre contenere tutti gli elementi che siamo intenzionati ad inserire.

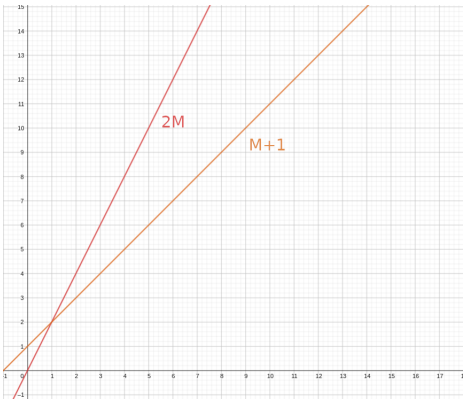


Figura 16: Dimensione soluzione vs spazio al crescere di M

In questo caso **Greedy-split** si dimostra perfettamente identico in termini di soluzione a **Greedy**, in quanto non si incontra mai un elemento che causa il superamento del limite c, unica differenza dei due algoritmi.

3.9 C.9

Illustrare gli algoritmi ***Greedy-Split*** *e* ***Ext-Greedy****. Discutere la qualità della soluzione prodotta da entrambi gli algoritmi, se applicati all'istanza* ***KP***

La qualità di tutti gli algoritmi greedy può essere migliorata ordinando gli elementi in base al rapporto $\frac{valore}{peso}$, operazione con costo $O(n \log_n)$. In generale le approssimazioni fornite da algoritmi greedy per la risoluzione di ***Knapspack*** sono molto vicine alla risposta. Per entrambi gli algoritmi presupponiamo tre strutture dati di dimensione n, "vettore" w dei pesi (che rappresenta l'isieme dei pesi degli elementi), "vettore" p dei profitti (che rappresenta l'isieme dei pesi degli elementi) e vettore x delle scelte. $x[j] \in 0, 1$ indica se l'elemento di indice j è inserito o no nello zaino. Inoltre supponiamo non esista alcun elemento di indice j tale che $p[j] > c$.

- **Greedy-split**:

```
w_tot = 0
z = 0
j = 0
while w_tot + w[j] <= c
                    x[j] = 1
                    w_tot = w_tot + w[j]
                    z = z + p[j]
                    j = j + 1
```

  Questo algoritmo interrompe la ricerca appena si incontra un elemento che causa il superamento del limite c.

  1. prova ad inserire un elemento di peso $w[j]$ nello zaino
  2. se l' aggiunta dell' elemento non causa il superamento del limite c inseriscilo, altrimenti concludi la visita.

Questo algoritmo, come **Greedy** soffre di un problema, è *potenzialmente pessimo*.
L' istanza di ***KP*** presentata nella domanda è un esempio di questa situazione:

$$(\vec{w} = (1, M), \vec{p} = (2, M), M)$$

Infatti ordinando gli elementi per $pw = \frac{profit}{weight}$ notiamo che

$$pw_1 > pw_2$$
$$\frac{2}{1} > \frac{M}{M} = 1$$

Seguendo l'algoritmo **Greedy-split** verrà scelto l'elemento con pw maggiore, cioè l'elemento con indice 1.
Finchè $M \leq p_1$ questo ci fornisce una *risposta* in quanto il scegliendo l'elemento con indice 1 abbiamo un profitto maggiore o uguale di quello che avremmo scegliendo l'elemento con indice 2 (quello con *peso* e *profitto* pari ad M.
Quando però $M > p_1$ l'algoritmo sceglie erroneamente, infatti considerando $M = 3$ il profitto ottenuto scegliendo l'elemento di indice 1 è 2, mentre il profitto ottenibile scegliendo il secondo elemento è 3.
Il rapporto $\frac{profitto\ ottenuto}{profitto\ ottimo}$ decresce al crescere di M rendendo sempre più "incorretta" la soluzione offerta da **Greedy-split**.

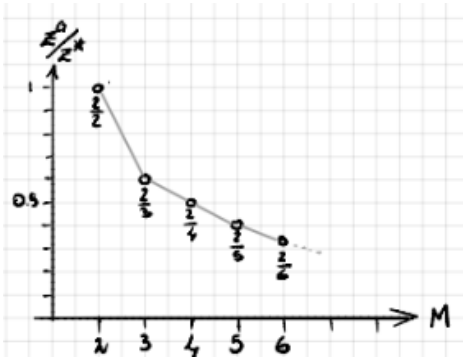


Figura 17: Qualità della soluzione di **Ext-Greedy** al crescere di M

- **Ext-Greedy**:

```
1  w_tot = 0
2  z=0
3  for j = 0 to n do
4          if w_tot + w[j] <= c then
5                  x[j] = 1
6                  w_tot = w_tot + w[j]
7                  z = z + p[j]
8          else x[j] = 0
9  for j = 0 to n do
10         if p[j] > z
```

```
11                  z = p[j]
12                  for k = 0 to n do x[k] = 0
13                  x[j] = 1
```

1. prova ad inserire un elemento di peso $w[j]$ nello zaino (riga 4)
2. se l' aggiunta dell' elemento non causa il superamento del limite c inseriscilo, passa al prossimo altrimenti. (righe 3-8)
3. analizzati tutti gli elementi concludi il ciclo.
4. confronta il valore ottenuto dal ciclo con il valore di ogni oggetto, se esiste un elemento che fornisce un risultato migliore di quello trovato nel ciclo (righe 3-8) allora inserisco solo quell'elemento nello zaino. $Z_{risposta} = max\{z, max\{p_1...p_n\}\}$

Questo algoritmo risolve il problema degli algoritmi **Greedy** e **Greedy-split** in qquanto se si verifica una situazione analoga a quella presentata nella domanda ($\vec{w} = (1, M), \vec{p} = (2, M), M$) verrà preso l'elemento avente peso e profitto M in quanto il suo profitto è superiore al profitto offerto dagli elementi inseriti in base all'incremento locale.

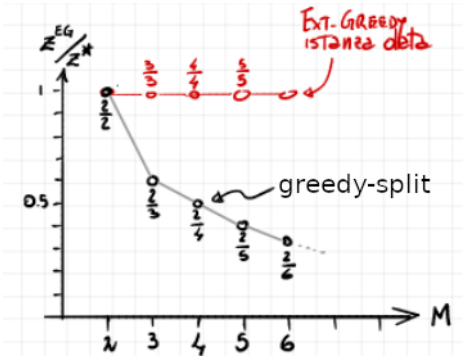


Figura 18: Comportamento **Ext-Greedy** rispetto a **Greedy-split**

3.10 C.10

*Illustrare l'algoritmo **Ext-Greedy**, inquadrare il significato della classe di algoritmi k-approximation e dimostrare che **Ext-Greedy** appartiene alla classe 1/2-approxiamtion. Argomentare sul perché **Ext-Greedy** non può appartenere a una classe di migliore di 1/2-approximation.*

Ext-Greedy:

```
1  w_tot = 0
2  z=0
3  for j = 0 to n do
4          if w_tot + w[j] <= c then
5                  x[j] = 1
6                  w_tot = w_tot + w[j]
```

```
 7                   z = z + p[j]
 8           else x[j] = 0
 9  for j = 0 to n do
10           if p[j] > z
11                   z = p[j]
12                   for k = 0 to n do x[k] = 0
13                   x[j] = 1
```

1. prova ad inserire un elemento di peso $w[j]$ nello zaino (riga 4)
2. se l' aggiunta dell' elemento non causa il superamento del limite c inseriscilo, passa al prossimo altrimenti. (righe 3-8)
3. analizzati tutti gli elementi concludi il ciclo.
4. confronta il valore ottenuto dal ciclo con il valore di ogni oggetto, se esiste un elemento che fornisce un risultato migliore di quello trovato nel ciclo (righe 3-8) allora inserisco solo quell'elemento nello zaino. $Z_{risposta} = max\{z, max\{p_1...p_n\}\}$

Definizione (K-approssimazione)**.** *:Un algoritmo G fonisce una k-approssimazione con* $0 \leq k \leq 1$ *se per ogni istanza I del problema P che G risolve , il rapporto tra il profitto calcolato con* $G, z^G(I)$ *ed il profitto ottimale* $z_p^*(I)$ *è almeno pari a k:*

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

La k-approximation è tight, stretta, se non è possibile migliorarla.

$$\exists T, \frac{z^G(T)}{z_p^*(T)} = k$$

Teorema. *Ext-greedy* $\in \frac{1}{2} - approximation$ *del* ***Knapsack problem***

$$\forall I_{kp}, \frac{z^{EG}(I_{kp})}{z_p^*(I_{kp})} \geq \frac{1}{2}$$

Dimostrazione:

La dimostrazione si basa sulla gerarchia dei profitti presentata in sezione 3.13.

$$\hat{p} \leq z^G \leq z^* \leq \lfloor z^L P \rfloor \leq z^L P \leq \hat{p} + p_s plit \leq z^G + p_{split}$$

$\forall I_{kp}\ z^*(I_{kp}) \leq z^G(I_{kp}) + p_{max}$ ove p_{max} è il massimo profitto offerto da un elemento (non è p-split).
possiamo maggiorare entrambi i membri dell'equazione:

$$z^G(I_{kp}) \leq max\{z^G(I_{kp}), p_{max}\}$$

$$p_{max} \leq max\{z^G(I_{kp}), p_{max}\}$$

$$allora\ z^G(I_{kp}) + p_{max} \leq max\{z^G(I_{kp}), p_{max}\} + max\{z^G(I_{kp}), p_{max}\}$$

Notiamo che l'espressione $max\{z^G(I_{kp}), p_{max}\}$ coincide con la definizione di **Extended-Greedy**, quindi posso sostituirlo nell'equazione.

$$\triangleq z^{EXT_GREEDY}(I_{kp}) + z^{EXT_GREEDY}(I_{kp}) = 2z^{EXT_GREEDY}(I_{kp})$$

nel caso peggiore il *profitto ottimale* è il doppio del profitto fornito da **Extended-Greedy**.

Teorema. $\frac{1}{2}$ *è una k-approssimazione tight per* ***Ext-Greedy***

$$\nexists K', K' > \frac{1}{2} \; t.c. \; \forall I, \frac{z^{EG}(I)}{z_p^*(I)} \geq K'$$

Dimostrazione:

Per dimostrarlo basta trovare un'istanza I che fornisca un rapporto $K = \frac{1}{2}$.In realtà possiamo identificare un'intera classe.

$$I = ((\vec{w} = (1, M, M), \vec{p} = (Q, M, M), 2M) \text{ con } 1 < Q < 2 \leq M$$

$$Z^{EG} = max\{Z^G(I), \; max\{Q, M\}\} = max\{Z^G(I), M\} = Q + M$$

la prima uguaglianza deriva dal fatto che $Q < M$, la seconda dal fatto che dato il rapporto $\frac{profit}{weight}$ **Greedy** seglierà sicuramente il primo elemento.Analizziamo gli estremi:

- $Q = 1 + \epsilon$

$$\lim_{M\to\infty} \frac{Z^{EG}(I)}{Z^*(I)} = \lim_{M\to\infty} \frac{1 + \epsilon + M}{2M} = \lim_{M\to\infty} \frac{1 + \epsilon}{2M} + frac12 = \frac{1}{2}$$

- $Q = 2 - \epsilon$

$$\lim_{M\to\infty} \frac{Z^{EG}(I)}{Z^*(I)} = \lim_{M\to\infty} \frac{2 - \epsilon + M}{2M} = \lim_{M\to\infty} \frac{2 - \epsilon}{2M} + frac12 = \frac{1}{2}$$

Dato che entrambi i limiti tendono a $\frac{1}{2}$ si può affermare che $\forall Q 1 + \epsilon \leq Q \leq 2 - \epsilon \Rightarrow Z^{EG} \in \frac{1}{2} - approx$

3.11 C.11

Illustrare l'algoritmo ***Greedy-LKP****, illustrandone il funzionamento su una istanza di* ***KP****, come, nell'esempio seguente, ed enunciarne la proprietà principale.*

$$(\vec{w} = (2, 3, 6, 7, 5, 4), \vec{p} = (6, 5, 8, 9, 6, 3), 10)$$

LKP è un rilassamento lineare del problema dello zaino, non è quindi necessario trovare una soluzione intera e si può formalizzare come il seguente problema di programmazione lineare:

$$\text{maximize} \sum_{k=1}^{n} p_k x_k$$

$$\text{subject to} \sum_{k=1}^{n} w_k x_k$$

$$x_1, ..., x_n \in [0, 1]$$

L'algoritmo di risoluzione **Greedy-LKP** ripercorre gli stessi passi dell'algoritmo **Greedy-split** ma quando incontra un elemento che causa il superamento del limite c al posto di interrompersi ne inserisce la massima frazione contenibile nello spazio rimasto:
Greedy-LKP:

```
1  w_tot = 0
2  z = 0
3  j = 0
4  while w_tot + w[j] <= c
5          x[j] = 1
6          w_tot = w_tot + w[j]
7          z = z + p[j]
8          j = j + 1
9  z_lp = z + ((c-w_tot)/w[j])* p[j]
10 x_lp[j] = ((c-w_tot)/w[j])
```

La sostanziale differenza sta nella riga 9,nel calcolo del profitto generato della frazione di oggetto inseribile nello zaino, qui riportato per chiarezza:

$$\frac{c - w_{tot}}{w_j} * p_j$$

Volendo mantenere anche l'informazione di quale combinazione di oggetti porta alla soluzione bisognerebbe trasformare la struttura dati $x^{LP} \in [0, .., 1]$ in modo da permettergli di contenere valori razionali e inserire all'indice j la frazione di oggetto: $x^{LP}[j] = \frac{c-w_{tot}}{w_j}$ (riga 10)

Greedy-LKP non fornisce una soluzione per il problema dello zaino, ma può essere utilizzato per fornire una stima per eccesso della *risposta* z^*.
Greedy-LKP genera una *risposta* ottimale, quindi siamo sicuri che nessun algoritmo per la soluzione di ***KP*** troverà una *soluzione* maggiore di quella ottenuta con **LKP**. Dire che un algoritmo gode della cosiddetta *greedy-choice property* vuol dire che "scelte ottime locali forniscono un ottimo locale".**Greedy-LKP** è un esempio di tale situazione. Eseguiamo **Greedy-LKP** sull'istanza del problema:

$$(\vec{w} = (2, 3, 6, 7, 5, 4), \vec{p} = (6, 5, 8, 9, 6, 3), 10)$$

Per prima cosa gli elementi vanno ordinati secondo $\frac{profit}{weight}$ decrescente, nel problema fornito sono già in questo ordine.

$$(\vec{w} = (2, 3, 6, 7, 5, 4), \vec{p} = (6, 5, 8, 9, 6, 3), 10)$$

Si prosegue poi come **Greedy-split**, iterando finchè non si supera $c = 10$

$$z.w = 2 + 3 + 6 = 11 > 10 \qquad z.p = 6 + 5 + 8 = 19$$

Considerando che l'ultimo elemento supera la capacità dello zaino si rimuove l'ultimo oggetto inserito e se ne prende la massima frazione

$$\frac{c - w_{tot}}{w_j} \rightarrow \frac{10 - 5}{6} = \frac{5}{6}$$

Ottenuta la frazione si calcola il *profitto* ed il corrispondente vettore risposta

$$z^{LP} = 6 + 5 + (8 * \frac{5}{6}) = \frac{53}{3} \qquad x^{LP} = (1, 1, \frac{5}{6}, 0, 0, 0)$$

3.12 C.12

*Dimostrare almeno un caso dell'enunciato: "**Greedy-LKP** gode della proprità Greedy Choice Property".*

Dire che un algoritmo gode della cosiddetta *greedy-choice property* vuol dire che "scelte ottime locali forniscono un ottimo locale".**Greedy-LKP** è un esempio di tale situazione.

Teorema. *Se* $\frac{p_1}{w_1} \leq \frac{p_2}{w_2} \leq ...$ *allora il vettore soluzione* $x^{LP} = (1, ..., 1, \frac{c - \sum_{1 \leq k \leq s} w_k}{w_s}, 0, ..., 0)$ *che produce* $z^{LP} = (\sum_{1 \leq k \leq s} p_k) + \frac{p_s}{w_s}(c - \sum_{1 \leq k \leq s} w_k)$ *è ottimale*

Per semplicità chiamiamo $c - \sum_{1 \leq k \leq s} w_k$ w_s^{LP} *Dimostrazione*:

Supponiamo per assurdo che esista un vettore soluzione migliore.

$$\exists y = (y_1, ..., y_n) \; t.c. \sum_{1 \leq k \leq n} y_n w_k > \sum_{1 \leq k \leq s} p_k + \frac{p_s}{w_s} w_s^{LP} \text{ e } \sum_{1 \leq k \leq n} y_n w_k = (\sum_{1 \leq k \leq s} w_k) + w_s^{LP}$$

Cioè un vettore y che fonisce un profitto migliore di x^{LP} saturando lo spazio.
Per proseguire la dimostrazione bisogna analizzare caso per caso in cosa è differente il vettore y dal vettore x^{LP}.
Analizziamo ad esempio il seguente caso, in cui il vettore y (sopra) è presentato a confronto con il vettore x^{LP} (sotto) $y_a w_a + y_b w_b = w_a$

1	y_a	1	x_s^{LP}	0	y_b	0
1	1	1	x_s^{LP}	0	0	0
	a		s		b	

dato che lo spazio disponibile deve essere riempito qualsiasi frazione di peso dell'elemento con indice a (che potrebbe anche essere 1, toglo tutte l'elemento) che io tolgo deve essere "riaggiunta" con l'inserimento di parte dell'elemento b.

$$y_b w_b = w_a(1 - y_a)$$

$$\frac{w_b}{w_a} = \frac{1 - y_a}{y_b}$$

Invece per ipotesi il profitto generato deve essere maggiore, quindi $y_a p_a + y_b p_b = p_a$

$$y_b p_b = p_a(1 - y_a)$$

$$\frac{p_b}{p_a} = \frac{1 - p_a}{p_b}$$

Ciò vuol dire che $\frac{p_b}{p_a} > \frac{w_b}{w_a}$ che può essere riscritto come $\frac{p_b}{w_b} > \frac{p_a}{w_a}$ È ciò va contro l'ipotesi, in quando il vettore è ordinato in base al rapporto $\frac{price}{weight}$.

3.13 C.13

Illustrare significato ed origine della sequenza di inclusioni tra profitti seguenti

La seguente gerarchia illustra i valori di profitto offerti dai vari algoritmi:

$$\hat{p} \leq z^G \leq z^* \leq \lfloor z^L P \rfloor \leq z^L P \leq \hat{p} + p_s split \leq z^G + p_{split}$$

Abbiamo i valori ottenuti da **Greedy-split** ($\hat{p}$), **Greedy**(z^G) e **Greedy-LKP**(z^{LP}). Aggiungendo invece p_{split} al risultato datomi da **Greedy** e da **Greedy-split** supero l'approssimazione datami da Z^{LP}, in quanto la parte di elemento aggiunta in **Greedy-LKP** è una frazione di $p_s split$, essendo una frazione è sicuramente minore.

Lo scopo di questa gerarchia di inclusioni è fornire approssimazioni per eccesso e per difetto ragionevolmente accurate del profitto ottimale z^*, con un basso costo computazionale, $O(n \log n)$. Queste approssimazioni servirano a fornire un *upper bound* ed un *lower bound* necessari alla generazione di un algoritmo **Branch&Bound** per la risoluzione di ***KP***.

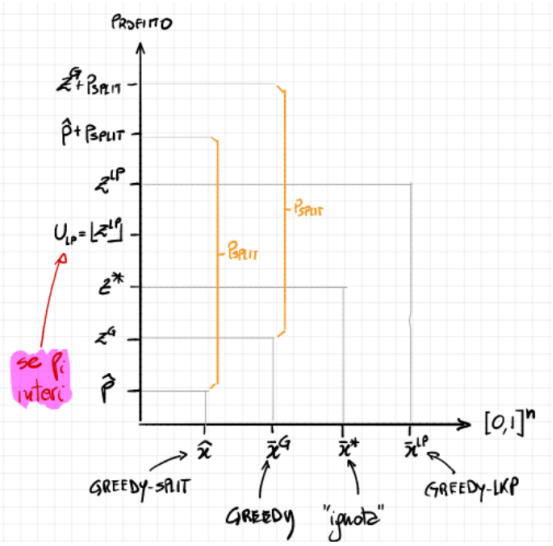


Figura 19: Grafico della gerarchia di inclusioni

3.14 C.14

Discutere la sintesi di un algoritmo Branch&Bound per KP

Nella creazione di un algoritmo *Branch&Bound* per *KP* la prima cosa da fare è stabilire una **funzione costo**.Questo può essere fatto basandosi sulla gerarchia dei profitti degli algoritmi greedy vista in precedenza.In particolare il costo ad un *E-nodex*$[0..j)$ può essere scritto come:

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

- Nella prima parte dell'equazione abbiamo il *costo noto*, che non è altro che la somma di tutti i profitti dei nodi "inseriti nello zaino" fin'ora ed i

loro pesi.

$$f(h(x[0..j)) \begin{cases} fh.w(x[0..j)) = \sum_{0 \leq k < j} x_k w_k \\ fh.z(x[0..j)) = \sum_{0 \leq k < j} x_k p_k \end{cases}$$

- La seconda parte di equazione fornisce una stima per eccesso del profitto ottenibile (*Upper bound*).

$$\hat{g}(x[0..j)) \begin{cases} \hat{g}.w(x[0..j)) = (\sum_{j \leq k < split} w_k) + (c - [fh.w(x[0..j)) + \sum_{j \leq k < split} w_k] \\ \hat{g}.z(x[0..j)) = (\sum_{j \leq k < split} p_k) + (c - [fh.w(x[0..j)) + \frac{\sum_{j \leq k < split} w_k]}{W_{split}} P_{split} \end{cases}$$

- Una terza misura utile a scrivere un algoritmo *Branch&Bound* per il problema Knapspack è una stima per difetto (*Lower Bound*), facilmente ottenibile tramite l'algoritmo greedy (o greedy split).

$$z^g(x[0..j)) = \sum_{0 \leq k < j} x_k p_k + \sum_{j \leq k < split} p_k + \sum_{split < k < n} x_k p_k$$
$$se \sum_{0 \leq k < j} x_k w_k + \sum_{j \leq k < split} w_k + \sum_{split < k < n} x_k w_k$$

Una volta descritte le 3 misure si può impostare un algoritmo per la risoluzione *Branch&Bound* che rispetti la seguente invariante: In qualsiasi istante della visita *lower bound* $< z^* <$ *upper bound*
L'idea di base è quindi quella di raffinare tramite le diverse iterazioni i due limiti fino ad ottenere la miglior *soluzione* che sarà la *risposta*.
Passo base:

Non avendo visitato alcun nodo l' *E-node* è la radice dello spazio degli stati. Il *lower bound* è quindi l'algoritmo greedy scelto applicato al problema completo, allo stesso modo l' *upper bound* è il risultato dell'algoritmo di soluzione di *LKP* applicato al problema completo. In questo stadio la miglior soluzione (che identifichiamo come $x[0..r)$) è la soluzione trovata tramite l'algoritmo greedy.

Passo induttivo:

Al passo induttivo l'attuale *E-node* sarà $x[0..j)$ e il nodo che fornisce la miglior soluzione $x[0..r)$ garantisce un profitto $z^*(x[0..r))$. Durante la visita dell' *E-node* si possono presentare 4 situazioni (non esclusive):

- $f(h(x[0..j)) > C$
  Non ha senso "espandere" il nodo e proseguire nella visita del sottoalbero $T[0..j)$ in quanto sicuramente non otterremo una risposta avendo superato la capacità massima.Il nodo viene etichettato come **"completo"**, si sta effettuando *pruning*.
- $f(h(x[0..j)) + \hat{g}.z(x[0..j)) < z^*(x[0..r))$
  Il profitto ottenibile espandendo il nodo è $<$ dell'approssimazione per difetto trovata in uno dei passi precendenti.Anche inserendo tutti gli

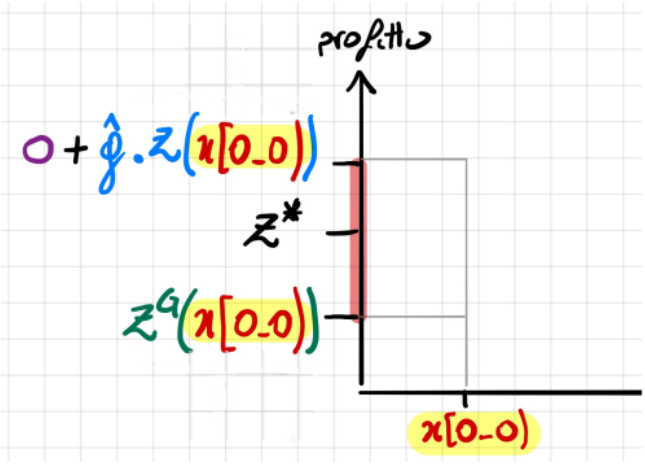


Figura 20: Bounds del profitto z

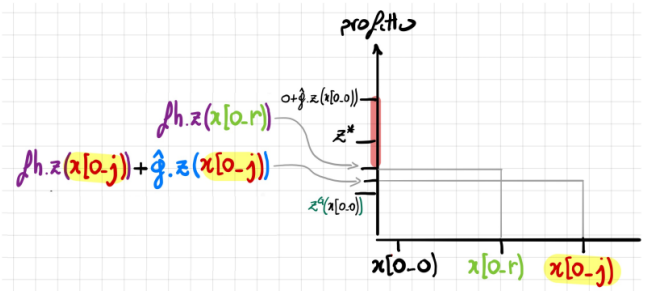


Figura 21: nessun miglioramento rispetto alla soluzione del nodo $x[0..r)$

oggetti fino a capienza massima non si troverebbe una soluzione migliore di quella attuale (ricordo essere $z^*(x[0..r))$).Il nodo viene etichettato come **"completo"**, si sta effettuando *pruning*.

- $f(h(x[0..j)) < C \wedge f(h(x[0..j)) + \hat{g}.z(x[0..j)) > z^*(x[0..r))$
  Il sottoalbero $T[0..j)$ può ancora generare *soluzioni*, fra queste potrebbe essere presente la *risposta*.E quindi necessario **espandere** il nodo e continuare l'esplorazione del sottoalbero.
- $f(h(x[0..j)) < C \wedge f(h(x[0..j)) > z^*(x[0..r))$
  E' stata trovata una *soluzione* che migliora il profitto $z^*(x[0..r))$, viene salvata ed i passi successivi faranno riferimento al profitto generato da $x[0..j)$.

La cosa non specificata dall'algoritmo è il criterio di scelta del prossimo *E-node* tra i vari *live-nodes* disponibili.Utilizzando un criterio FIFO si ottengono buoni risultati, ma usando una tecnica *least-cost* si è scoperto sperimentalmente che le prestazioni sono migliori. Intuitivamente ci sono due motivi per questo: in primis se $z^{LP} = Z^*$ è possibile che venga trovata una soluzione che sappiamo con certezza essere anche *risposta*, si può quindi

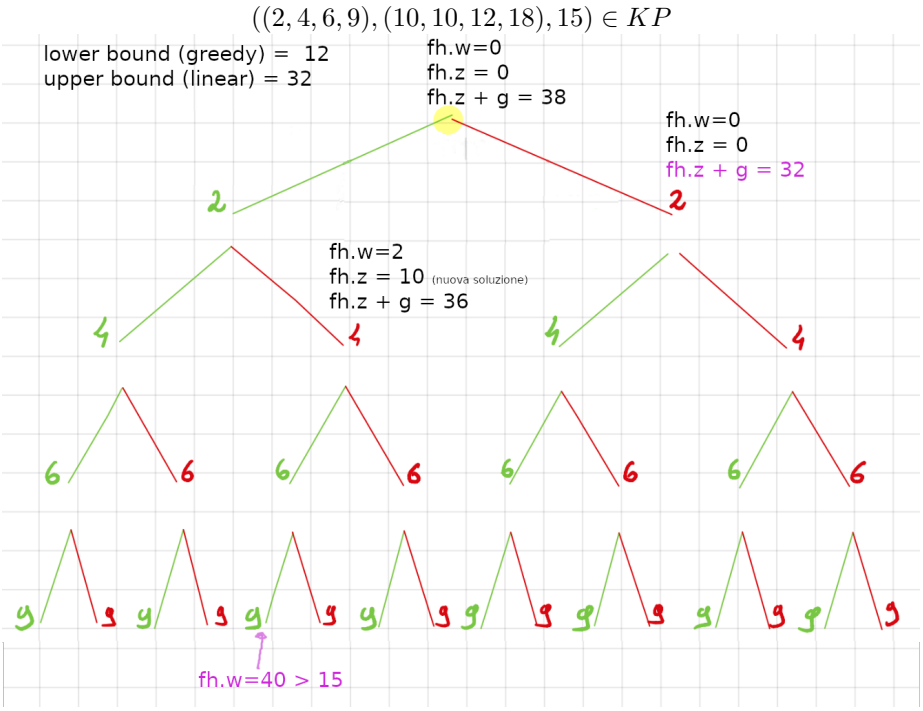


Figura 22: Esempio di Branch&Bound

fare pruning dei restanti rami. In secondo luogo è possibile che giungendo "prima" alle *soluzioni* migliori il valore di $z^*(x[0..r))$ cresca rapidamente causando il "rifiuto" di nodi che con una politica FIFO sarebbero stati espansi.(I nodi vengono rifiutati per la situazione 2)

4 Programmazione dinamica

4.1 D.1

emphIllustrare una soluzione ricorsiva del problema **KP**, impostata in accordo con la tecnica "Programmazione Dinamica" con accenni alla sua complessità pseudo-polinomiale.

Il *problema dello zaino* può essere suddifiso in una serie di sotto-problemi **KP** tra di loro indipendenti, prendiamo ad esempio $KP = (\vec{w} = (3,5,9,4), \vec{p} = (5,6,7,3), 9)$
Questo problema può essere visto secondo il seguente formalismo:

$$max \sum_{j=1}^{4} p_j x_j \text{ con } \sum_{j=1}^{4} w_j x_j$$

Estraendo l'elemento di indice 4 si ottiene: $max\ 3x_4 \sum_{j=1}^{4} p_j x_j$ con $4x_4 \sum_{j=1}^{4} w_j x_j$.
In tal modo si può generare l'albero delle scelte seguendo il formalismo, ad esempio la scelta $x = [?,1,0,1]$ (scelta incompleta, non è una *soluzione*) può essere

formalizzata in: $max\ 3+6+5x_1 \sum_{j=1}^{0} p_j x_j$ con $4x_4 \sum_{j=1}^{4} w_j x_j$.
Sviluppando completamente l'albero delle scelte come in figura 23 si nota come l'istanza del problema **KP** presenta dei sottoproblemi condivisi, si può quindi configurare una soluzione che faccia uso delle tecniche di *programmazione dinamica*.

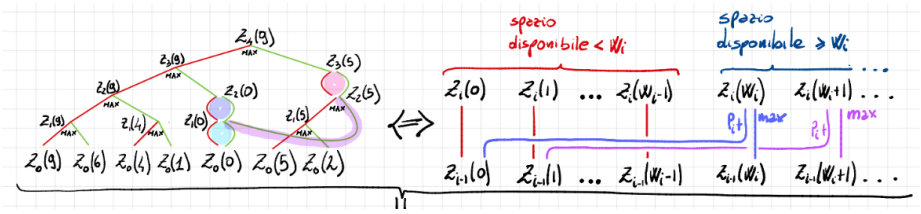


Figura 23: Albero dei sottoproblemi

Volendo inserire l'elemento i-esimo abbiamo due possibili situazioni:

- $W_i >$ spazio disponibile.Quindi w_i non ci sta, la soluzione è identica alla soluzione del problema avente stesso spazio disponibile ma senza w_i.
- $W_i <$ spazio disponibile.In questo caso posso inserire w_i, ed il profitto generato sarà il massimo tra: la soluzione del problema senza w_i e con spazio residuo "spazio disponibile - w_i + il profitto generato da i e la soluzione del problema senza w_i con spazio residuo "spazio disponibile" Scritto in maniera più chiara e più formale:

  $$Z_i(W_i + sd) = \max\{Z_{i-1}(sd - w_i) + P_i, Z_{i-1}(sd)\}$$

  dove sd è lo spazio disponibile. Intuitivamente nel primo caso abbiamo inserito nello zaino i quindi il profitto è aumentato di p_i e la dimensione diminuita di w_i, nel secondo caso non lo abbiamo aggiunto, quindi la soluzione ha lo stesso profitto della soluzione precedente.

Le due situazioni possono essere riassunte e formalizzate in questa formula ricorsiva:

$$z_j(sd) = \begin{cases} z_{j-1}(sd) \\ \max\{Z_{i-1}(sd - w_j) + P_j, Z_{j-1}(sd)\} \end{cases}$$

Seguendo questa definizione ricorsiva si può ricavare un algoritmo che genera una "tabella dei profitti" completa.

Verrebbe da dire che la complessità di questo algoritmo è $O(n * c)$, ma la complessità va rapportata alla rappresentazione più efficiente dell'input. Rispetto alla dimensione dell'input composto dai pesi $w_1, ..., w_n$ e dei profitti $w_1, ..., w_n$ ($|\vec{w}| = |\vec{p}| = n$) la complessità è polinomiale.Il singolo peso (o profitto) è rappresentato tramite una notazione posizionale (probabilmente binario) e ci vogliono n istanze di quella notazione per rappresentare tutti i pesi (o profitti).
Per c la complessità non è polinomiale però. c è un numero, ed in quanto numero è rappresentato da 2^b bit (presuppongo rappresentazione binaria). Quindi al crescere di c la sua rappresentazione cresce seguendo l'equazione $bits = 2^{1+\lfloor \log_2 c \rfloor}$ e ciò fa si che la complessità sia esponenziale rispetto a c.

La complessità dell'algoritmo di *programmazione dinamica* per la risoluzione di **KP** è ***Pseudo polinomiale*** in quanto è polinomiale per almeno uno dei valori dell'input (c).

4.2 D.2

*Illustrare vari algoritmi iterativi che implementano soluzioni al problema **KP**, impostati in accordo con la tecnica "Programmazione Dinamica".*

L'algoritmo ricorsivo sviluppato tramite *programmazione dinamica*permette di costruire una soluzione generando una tabella la cui ispirazione può essere ricondotta alla figura 23.

$Z_n(0)$	$Z_n(1)$	...	$Z_n(c)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$Z_1(0)$	$Z_1(1)$	...	$Z_0(c)$
$Z_0(0)$	$Z_0(1)$	...	$Z_0(c)$

Dall'idea di questa tabella il più semplice algoritmo iterativo che ne deriva è:

```
1  for d = 0 to c
2      z[0][d] = 0
3  for j = 1 to n do
4      for d = 0 to w[j] -1 do
5          z[j][d] = z[j-1][d]
6      for d = w[j] to c do
7          if z[j-1][d-w[j]] + p[j] > z[j-1][d]
8              z[j][d]=z[j-1][d-w[j]] + p[j]
9              A[j][d]=1
10         else
11             z[j][d] = z[j-1][d]
12             A[j][d]=0
13 z_sol=z[n][c]
```

- nelle righe 1-2 vi è l'inizializzazione, la prima colonna $z_0(0), ..., z_n(0)$ è impostata a 0 dato che non c'è spazio nello zaino per inserire nessun elemento.

- Le righe 4-5 illustrano la situazione "j è troppo grande per essere inserito", quindi viene copiato il valore del passo precedente

- Dalla riga 6 alla riga 12 abbiamo l'implementazione dell'eq ricorsiva

$$z_j(sd) = \begin{cases} z_{j-1}(sd) \\ \max\{Z_{j-1}(sd - w_j) + P_i, Z_{j-1}(sd)\} \end{cases}$$

  Nelle righe 7-9 abbiamo il caso in cui j migliora la soluzione, quindi la soluzione viene aggiornata aggiungendo j, nelle righe 11-12 invece abbiamo la situazione in cui è meglio mantere la vecchia soluzione dato che questa fornisce un profito migliore.

Questo algoritmo si porta dietro una struttura dati aggiuntiva, la tabella A che mantiene l'informazione di quale sia la tupla che risolve il problema (la *risposta*).

Una prima ottimizzazione sullo spazio si può fare liberandosi di questa matrice e ricavando la tupla *risposta* dalla tabella z.

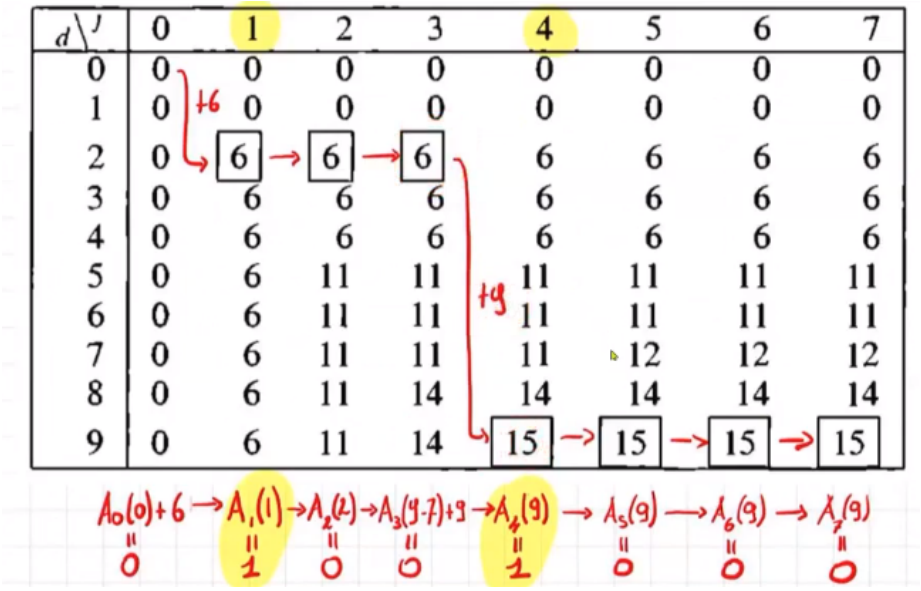


Figura 24: Ricostruzione della tupla risolvente

Partendo dal valore $z_n(c)$ si ripercorre "all'indietro" l'algoritmo, se non vi è una variazione di valore è perchè si è preso il secondo ramo dell'if (riga 10), se vi è una variazione è perchè e stato aggiunto un elemento allo zaino.Bisogna quindi individuare la colonna in cui tale variazione è avvenuta e segnarsi l'indice della stessa (dato che gli indici delle colonne corrispondono agli indice del vettore di elementi x).
Ma come ricavare $z_{j-1}(d)$ in modo da sapere con quale "riga proseguire"? Basta rimuovere il profitto dell'elemento j dalla cella $z_j(d)$ per individuare quale fosse il valore prima dell'aggiunta, cercare il valore corrispondente sulla $j-1$.

Si può ottimizzare ancora andando a "ridurre" la tabella a due soli array, uno contenente i valori della $i-1$-esima colonna e l'altro contenente i valori della colonna con indice i, attualmente in costruzione.
Una volta completato il calcolo della colonna i questa diventerà la colonna $i-1$ e verrà usata per generare la colonna successiva, fino alla colonna c.
Ma si può ancora proseguire accorpando i due vettori, infatti non è necessario mantenere l'informazione delle celle che non cambiano valore.Basta calcolare i nuovi valori usando i valori scritti nel resto del vettore avendo cura di aggiornare il vettore partendo "dal fondo".
Le due ottimizzazioni sono rappresentate in figura 25, e notiamo che effettuandole si perde però la possibilità di ricavarsi la tupla soluzione x partendo dalla

tabella. Notiamo anche che tutte queste ottimizzazioni sono relative alla memoria, non al tempo, che resta sempre $O(n * c)$

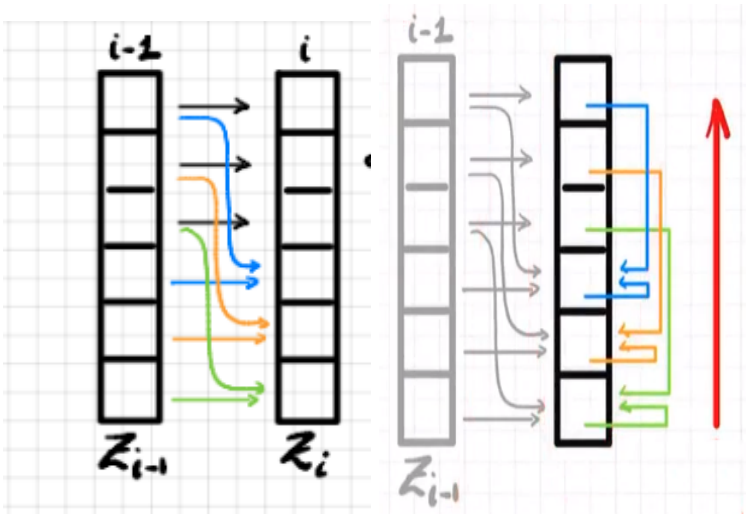


Figura 25: Ottimizzazioni della tabella

5 Complessità computazionale

5.1 E.1

*Illustrare le classi **PTime** ed **NPTime** in termini classici.*

Una volta accordatici su una ragionevole rappresentazione dell'input si può parlare dei modelli di computazione, riconducendoci alla capacità degli automi (Macchine di Turing) di riconoscere un linguaggio.
Le Macchine di Turing deterministiche (**DTM**) sono composte da un nastro infinito ed una testina in grado di leggere e scrivere, in particolare hanno 3 caratteristiche:

1. un set finito di simboli scrivibili sul nastro (alfabeto).
2. un set di stati in cui sono presenti alcuni stati particolari: uno *stato iniziale* ed uno o più *stati finali*.
3. una fuzione di transizione che deterministicamente spiega come si muove la testina.Deterministicamente poichè dato uno stato ed un simbolo in input segue una sola possibile configurazione futura.

Posta una **DTM** si può informalmente definite la complessità come *il numero di "passi" necessari alla computazione prima di giungere in uno stato finale*, naturalmente in realazione alla lunghezza dell'input.
La classe **PTime** contiene tutti i problemi che possono essere risolti tramite

una **DTM** in un numero di "passi" polinomiale.Formalmente:

$$\exists p \; t.c. \; \forall x \in \{0,1\}^* \; costo(DTM(x)) \leq p(|x|)$$

Cioè esiste un polinomio p tale che per ogni possibile sequenza di 0 ed 1 in input ha un costo computazionale minore o uguale del calcolo del polinomio sulla dimensione della sequenza di input.

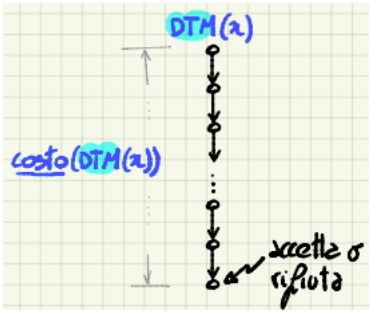


Figura 26: costo di DTM(x)

La classe **NPTime** si basa invece sul concetto di Macchina di Turing **non** deterministica (**NDTM**).
A differenza delle **DTM** non vi è una sola "testina", bensì due: un *controllo a stati finiti* che può leggere o scrivere (è la testina della **DTM**), limitato a lavorare sulle celle$[0 + \infty)$, ed un *guessing module* che lavora invece sulle celle $[-1, -\infty)$.
Questo "oracolo" scrive a caso nelle celle, generando una sequenza casuale di simboli dell'alfabeto (*guessing stage*. Anche la lunghezza della stringa generata è casuale, potenzialmente infinita.
Una volta concluso il lavoro del *guessing module* il *controllo a stati finiti* inizia a computare non solo basandosi sull'input e sullo stato, ma anche sulla stringa generata dall' *oracolo* in maniera non predicibile *checking stage*. Una stringa è accettata se il *controllo a stati finiti* si ferma su uno stato di accettazione, per una qualche stringa generata dal *guessing module*, se non vi è alcuna stringa "oracolare" che permette di arrivare ad uno stato di accettazione la stringa è rifiutata.

La classe **NPTime** contiene tutti i problemi che possono essere risolti tramite una **NDTM** in un numero di "passi" polinomiale.Formalmente:

$$\exists p \; t.c. \; \forall x \in \{0,1\}^* \; costo(NDTM(x)) \leq p(|x|)$$

È possibile vedere questa classe di algoritmi anche sfruttando il formalismo delle **DTM**

$$\exists p \; t.c. \; \forall x \in \{0,1\}^* \; costo(DTM(x,y)) \leq p(|x|)$$

Cioè la sequenza in input viene verificata in tempo polinomiale rispetto alla dimensione dell'input da una **DTM** a cui viene fornita non solo l'input ma anche la stringa generata dall'oracolo che si sà condurre ad una soluzione (comunenmente detta certificato).

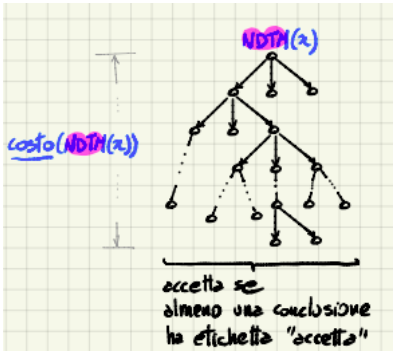


Figura 27: costo di NDTM(x)

5.2 E.2

*Illustrare le classi **PTime** ed **NPTime** in termini di possibili linguaggi imperativi paradigmatici.*

Utlizzare il formalismo delle macchine di Turing è scomodo e complesso per essere usato direttamente per il calcolo del costo computazionale. Abbiamo bisogno di lavorare su strutture dati complesse con strumenti più "di alto livello" senza doverci per forza ricondurre al binario, ma mantenendo la possibilità di definire la complessità.
La soluzione è definire dei linguaggi in cui poter scrivere i programmi che corrispondano ad una macchina di Turing, semplificandone l'utilizzo, come se fosse una API per interagire con una macchina di Turing.

- LID (*Linguaggio Imperativo Deterministico*) corrispondente alle **DTM**
  Possiamo quindi dire che la classe **PTime** è l' insieme dei problemi risolvibili da programmi scritti in un linguaggio imperativo, deterministico .
  Questo linguaggio è la base di ogni linguaggio di programmazione, dato che fornisce le operazioni di:
  - assegnazione: la possibilità di assegnare a variabili valori calcolati con espressioni aritmetiche.
  - sequenza: possibilità di decidere in che ordine verranno eseguite le assegnazioni
  - selezione: decidere in funzione del valore di una variabile quale operazione eseguire
  - iterazione: ripetere più volte una operazione.
- LNID (*Linguaggio Non Imperativo Deterministico*) corrispondente alle **NDTM**
  Questo linguaggio estende il LID aggiungendo una operazione *scelta(n)*. Questa primitiva genera n "branch" paralleli che proseguono l'esecuzione. Ad esempio l'istruzione `x <- scelta(3)` produce 3 "thread" che eseguono la prossima istruzione uno usando $x = 0$, un altro usando $x = 1$ e l'ultimo usando $x = 2$.

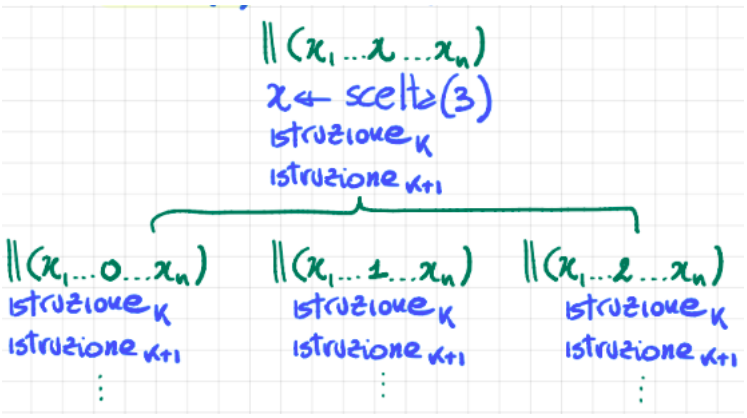


Figura 28: Istruzione scelta

In tal modo è possibile generare una programma che contemporaneamente computa tutti it possibili stati dell'albero delle scelte, come mostrato in figura 27

5.3 E.3

Argomentare sul perché la definizione della classe ***PTime*** *può essere basata indifferentemente sulla definizione di* ***Macchine di Turing*** *o su un ragionevole linguaggio di programmazione paradigmatico.*

Utlizzare il formalismo delle macchine di Turing è scomodo e complesso per essere usato direttamente per il calcolo del costo computazionale. Abbiamo bisogno di lavorare su strutture dati complesse con strumenti più "di alto livello" senza doverci per forza ricondurre al binario, ma mantenendo la possibilità di definire la complessità.
La soluzione è definire dei linguaggi in cui poter scrivere i programmi che corrispondano ad una macchina di Turing, semplificandone l'utilizzo, come se fosse una API per interagire con una macchina di Turing.

È possibile basare la definizione di classe **PTime** su un linguaggio **LID** (e quella di **NPTime** su un linguaggio **LIND**) perchè esiste una codifica tra il linguaggio e la macchina che non introduce complessità computazionale maggiore di un polinomio. Cioè è possibile "simulare" il funzionamento dell'algoritmo tramite una macchina di Turing e il costo di questa *simulazione* è polinomiale.
La figura 29 mostra che è possibile simulare un passo di un algoritmo, scritto in un linguaggio, facendo compiere una serie di passi ad una macchina di Turing.Posto di voler simulare c passi dell'algoritmo la corrispondente macchina necessiterà di $s(c)$ passi, ove la funzione s è polinomiale. Perciò possiamo affermare che un algoritmo scritto in linguaggio **LID** ha una data complessità perchè è possibile simulare il suo funzionamento tramite una macchina di Turing inserendo un overhead al massimo polinomiale, quindi senza modificare la sua complessità.

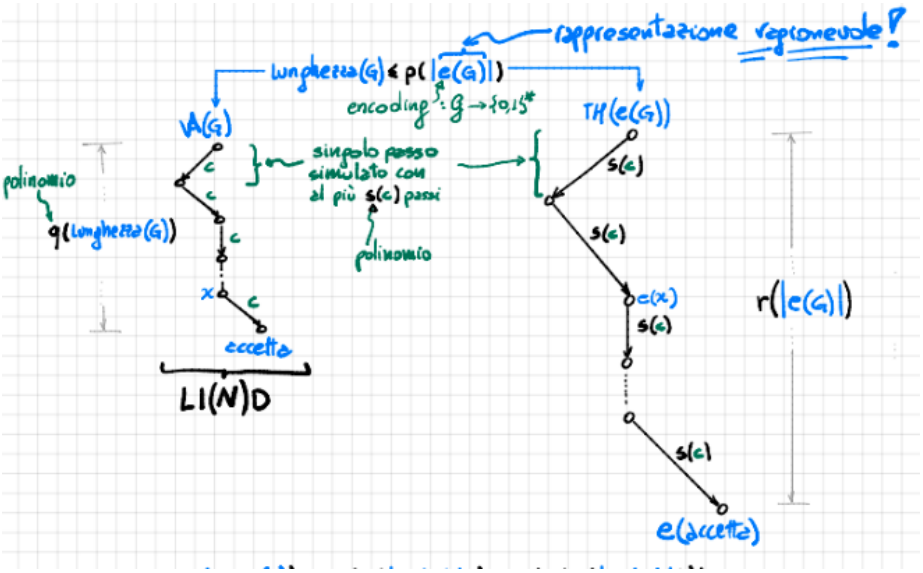


Figura 29: Overhead polinomiale

5.4 E.4

Argomentare sul perché proprietà sulla complessità computazionale di un problema di ottimizzazione sono trasferibili al corrispondente problema decisionale e viceversa.

La teoria della ***NP-completeness*** si è sviluppata in relazione alla complessità di problemi decisionali, non di ottimizzazione, a causa di come si è evoluta la teoria degli automi.
Infatti utilizzando il formalismo degli automi (Macchine di Turing) risulta molto più naturale guardare al problema come se fosse un linguaggio da decidere. Se l'input fornito appartiene al linguaggio l'automa risponderà *True*, *False* altrimenti.
Nasce quindi la necessità di passare da problemi di ottimizzazione a problemi decisionali, per poterne valutare la complessità, e viceversa.

Analizziamo il caso del problema **KP** per spiegare informalmente che questa trasformazione è sempre possibile.

- Ottimizzazione → Decisione, risolvere il problema decisionale vuol dire risolvere anche il corrispettivo problema di ottimizzazione.
  Dato un algoritmo che risolve **KP(l,c)** possiamo costruire un algoritmo per risolvere **KP(c)**, l'intuizione è sollevare il *lower bound l* finchè non trovo più alcuna soluzione. Segue l'algoritmo in pseudocodice in cui l'operazione $sum(p\_i)$ indica la somma di tutti i pesi degli elementi del vettore soluzione trovato.

```
l=0
z*=0
for (l=0 ; l <= sum(p_i), l++)
```

```
        if (KP(l, c) != true)
            z* = l-1
    return z*
```

- Decisione → Ottimizzazione, risolvere il problema di ottimizzazione vuol dire risolvere anche il corrispettivo problema decisionale.
  Nel caso del problema **Knapsack** è molto semplice da dimostrare.Avendo un algoritmo che risolve **KP(c)** possiamo ottenere una soluzione z^*, se tale soluzione è $\leq l$ rispondiamo $True$ al problema decisionale, $False$ altrimenti:

```
z* = KP(c)
if (z >= l)
    return true
else
    return false
```

5.5 E.5

*Ricordando il concetto "Riduzione", dimostrare che **KP** è intrattabile, assumendo che **SAT** lo sia.*

La *riduzione* tra problemi decisionali è un'operazione che permette di confrontare due problemi computazionali e la complessità delle loro soluzioni.
Si indica con "$\leq_p$", quindi $P_1 \leq_p P_2$ significa affermare che se P_2 si può risolvere "velocemente" allora anche P_1 può essere risolto velocemente. Negando l'implicazione se non è noto alcun algoritmo polinomiale per risolvere P_2 nemmeno P_1 può essere risolto in tempo polinomiale. Formalmente:

Definizione (Riduzione)**.** *Dati P_1 e P_2 problemi computazionali. $P_1 \leq_p P_2$ se $\exists F : Dom(P_1) \to Dom(P_2)$ con le seguenti proprietà:*

- *se $x \in Dom(P_3)$ ed x istanza di $P_1 \Rightarrow F(x)$ istanza di P_2, cioè se l'algoritmo che risolve il problema decisionale P_1 ritorna vero quando viene data in input l'istanza x allora ritornerà vero l'algoritmo che risolve P_2 data in input l'istanza $F(x)$.*
- *se $x \in Dom(P_1)$ ed x non è istanza di $P_1 \Rightarrow F(x)$ istanza di P_2, cioè se l'algoritmo che risolve il problema decisionale P_1 ritorna falso quando viene data in input l'istanza x allora ritornerà falso l'algoritmo che risolve P_2 data in input l'istanza $F(x)$.*
- *$\exists p, \forall x$ t.c. $costo(Alg_F(x)) \leq p(|x|)$, cioè il costo di applicare la trasformazione F all'istanza x è inferiore ad un polinomio calcolato sulla dimensione di x.*

Posto **SAT** intrattabile si può affermare che **KP** è intrattabile perchè $SAT \leq_p KP$, dalla seguente catena di riduzioni:

$$SAT \leq_p NNF \leq_p CNF \leq_p 3CNF \leq_p 3COL \leq_p EXCO \leq_p SubsetsumKP$$

- **SAT**: data una formula proposizionale f, dice se questa è soddisfacibile. $\exists \varphi \Phi_\varphi(f) = true$

- **NNF**: data una formula Negation Normal Form f, dice se esiste una assegnazione delle variabili $FV(f)$ che soddisfa f.
  $\exists \varphi : FV(f) \in \{true, false\}^* \; t.c. \; \Phi_\varphi(f) = true$
- **CNF**: data una formula f scritta in Conjunctive Normal Form, dice se questa è soddisfacibile. $\exists \varphi \Phi_\varphi(f) = true$.
  $\exists \varphi : FV(f) \in \{true, false\}^* \; t.c. \; \Phi_\varphi(f) = true$
- **3CNF**: data una formula f scritta in Conjunctive Normal Form a 3 letterali, dice se questa è soddisfacibile. $\exists \varphi \Phi_\varphi(f) = true$.
  $\exists \varphi : FV(f) \in \{true, false\}^* \; t.c. \; \Phi_\varphi(f) = true$
- **3COL**: Sia $G = (V, E)$, grafo non orientato $\exists c : Vertex \to col1, col2, col3$ t.c. $\forall u, v \in V \; e \; (u, v) \in E c(v) \neq c(u)$. Cioè data una funzione di etichettatura c che assegna un "colore" ad ogni nodo esiste modo di colorare il grafo senza che nodi adiacenti abbiano lo stesso colore.
- **EXCO**: Dati U ed $S = \mathcal{P}(U)$ esiste un sottoinsieme $S' \subseteq S$ che è una copertura esatta di U cioè tra gli insiemi elementi di S compaiono tutti ed una sola volta gli elementi di U:
  $$\forall z, y \in S' \; x \neq y \quad x \cap y = \emptyset \quad \bigcup_{x \in S'} X$$
- **SUBSETSUM**: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.

5.6 E.6

Illustrare un qualche aspetto fondamentale della dimostrazione che ***SAT*** $\leq_P$ ***NNF***.

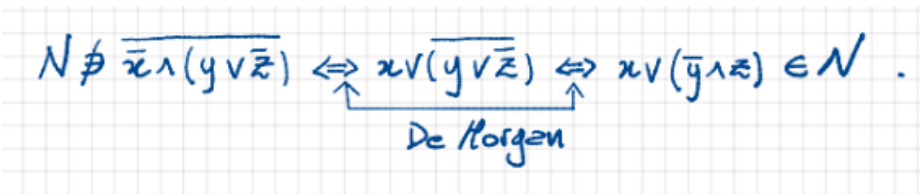
Le formule **NNF**(Negation Normal Form) sono tutte quelle formule in cui la negazione è presente solo sullevariabili, non sulle sotto-formule. La sintassi diventa quindi:

$$N ::= true | false | var | \overline{var} | N \wedge N | N \vee N$$

La *funzione di riduzione* che permette di passare da formule proposozionali a NNF è ben conosciuta, solo le leggi di De Morgan:

$$\overline{f \vee h} = \overline{f} \wedge \overline{h} \qquad \overline{f \wedge h} = \overline{f} \vee \overline{h}$$

L'intuizione è quella di "spingere" le negazioni presenti nella *formula proposizionale* f verso le variabili.



5.7 E.7

Nell'ambito della dimostrazione di ***NNF*** $\leq_P$ ***CNF****, illustrare intuizione sulla e definizione della trasformazione di Tseytin, dimostrando la sua proprietà principale.*

Le formule in **CNF** (Conjunctive Normal Form) sono formule composte da una congiunzione di clausole a loro volta composte da una disgiunzione di letterali.

Definizione (Sintassi)**.**

$$F ::= [C]|[C], F \; congiunzione \; (\wedge)$$

$$C ::= true|false|l, C \; disgiunzione \; (\vee)$$

Si può passare dalla forma in **NNF** alla forma in **CNF** grazie alla *trasformazione di Tseytin.*
Intuitivamente l'idea è quella di somporre l'equazione in tutte le sue sottoparti, collegate tra loro da "collegamenti" che corrispondono alle equivalenze logiche. Assegnando un nome ad ogni canale è possibile ricostruire la formula logica sotto forma di **NNF**, il modo più efficace di vedere questa equivalenza è un "circuito". Analizzando ad esempio $[(\overline{s} \wedge p) \vee ((q \wedge \overline{r}) \wedge p)]$

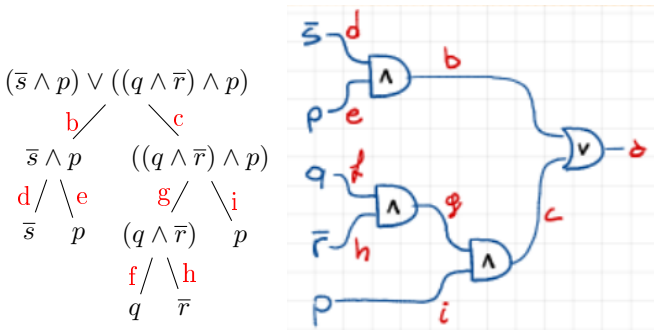


Figura 30: Trasformazione di Tseytin

Ricomponendo si ottiene:

$$a, (a \Leftrightarrow b \vee c), (b \Leftrightarrow d \wedge e), (c \Leftrightarrow g \wedge i), (g \Leftrightarrow f \wedge h)$$

. Eliminando le implicazioni si può ridurre la formula ad una formula avente solo congiunzioni (*and*) fuori dalle parentesi e disgiunzioni (*or*) dentro. La trasformazione di *Tseitin* può essere formalmente definita come:

Definizione. *(Trasformazione di Tseitin) Data f funzione in Negation Normal Form, la trasformazione di Tseitin T è:*

$$T(f) = [a], C[a, f]$$

Ove a è una nuova variabile, C è una operazione del tipo $C : variabili \times NNF$

$$C[a, l] = a \Leftrightarrow l$$

$$C[a, f \circ g] = a \Leftrightarrow b \circ c, C[b, f], C[c, g] \; con \circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$$

b e c nuove variabili.

La *Trasformazione di Tseitin* genera delle **CNF**, ma ciò va dimostrato:

Teorema (Correttezza trasformazione di Tseitin)**.** $\forall f$ *formula in* ***NNF*** *e* $\forall a$ *variabile* $C(a,f) = c_1, ..., c_n$ *sono tutte clausole.*

Dimostrazione:

Dimostrato per induzione su f, ci sono due possibilità:

- f è un letterale l:

  $C(a,l) = (a \Leftrightarrow l) = (a \Rightarrow l) \wedge (l \Rightarrow a) =$ sostituendo l'implicazione $= [\overline{a}, l], [\overline{l}, a]$

  Cioè due clausole.

- $f = g \circ h$, cioè f composizione di due operatori.

  $C(a,l) = a \Leftrightarrow (b \circ c), C(b,g), C(c,h)$ per definizione della trasformazione
  $a \Leftrightarrow (b \circ c), (C_1, ..., C_n), (C'_1, ..., C'_n)$ per ipotesi induttiva le due parentesi sono clausole
  $=$ sostituendo le implicazioni $= \overline{a} \vee (b \circ c), \overline{(b \circ c)} \vee a, (C_1, ..., C_n), (C'_1, ..., C'_n)$

  Tutte e quattro sono clausole.

5.8 E:8

Dimostrare ***NNF*** $\leq_P$ ***CNF****, supponendo di avere a disposizione la trasformazione di Tseytin ed il suo lemma principale.*

La dimostrazione di $NNF \leq_p CNF$ si basa su un lemma derivante dalla costruzione della *funzione di Tseitin*

$$\text{Lemma T} \begin{cases} (\exists \varphi \Phi_\varphi(f) = true) \Leftrightarrow (\exists \varphi' \Phi_{\varphi'}(C[a,f]) = true \wedge \varphi'(a) = true) \\ (\exists \varphi \Phi_\varphi(f) = false) \Leftrightarrow (\exists \varphi' \Phi_{\varphi'}(C[a,f]) = true \wedge \varphi'(a) = false) \end{cases}$$

Teorema. $f \in NNF \Leftrightarrow T(f) \in CNF$

Dimostrazione:

Per definizione $T(f) = [a], C[a,f]$.

- se f è soddisfacibile la sua trasformazione $T(f)$ è soddisfacibile, cioè
  $f \in NNF \Rightarrow \exists \varphi \Phi_\varphi(f) = true$
  Ci appoggiamo al Lemma T.1:

  $$(\exists \varphi \Phi_\varphi(f) = true) \Leftrightarrow (\exists \varphi' \Phi_{\varphi'}(C[a,f]) = true \wedge \varphi'(a) = true)$$

  Cioè esiste se esiste una funzione φ che assegna dei valori alle variabili di f tali che rendono vera $\Phi(f)$ allora esiste un'altra funzione φ' che applicata a $C[a,f]$ rende vera $\Phi(C[a,f]$ ed assegna il valore *vero* alla nuova variabile $a.\varphi'$ è una "estensione" di φ sulle nuove variabili generate da C che comunque "rispetta" il comportamento di φ.
  Posto vero per ipotesi $\exists \varphi \Phi_\varphi(f) = true$ ho che $(\varphi'(a) \wedge \varphi' \Phi_{\varphi'}(C[a,f])i) = true$.

Inserendo $\varphi'(a)$ come congiunzione all'interno della funzione di valutazione $\Phi_{\varphi'}$ si ottiene $\Phi_{\varphi'}([a], C[a, f])$ che riprendendo la definizione di T è $\Phi_{\varphi'}(T(f))$.Quindi$\Phi_{\varphi'}(T(f)) = true$, come volevasi dimostrare.

- se invece f non è soddisfacibili la sua trasformazione $T(f)$ non deve essere soddisfacibile, cioè $f \notin NNF \Rightarrow \forall\varphi\Phi_{\varphi}(f) = false$
  Appoggiandosi al lemma T.2 abbiamo che

  $$(\exists\varphi\Phi_{\varphi}(f) = false) \Leftrightarrow (\exists\varphi'\Phi_{\varphi'}(C[a, f]) = true \wedge \varphi'(a) = false)$$

  Dato che abbiamo supposto per ipotesi $\forall\varphi\Phi_{\varphi}(f) = false$ allora $(\varphi'(a) \wedge \varphi'\Phi_{\varphi'}(C[a, f])i) = false$.
  Il Lemma T.2 ci suggerisce che il valore falso deriva dalla congiunzione $true \wedge false = false$.
  Come nel primo caso inserendo $\varphi'(a)$ come congiunzione all'interno della funzione di valutazione $\Phi_{\varphi'}$ si ottiene $\Phi_{\varphi'}([a], C[a, f])$.Cioè $\Phi_{\varphi'}(T(f)) = \Phi_{\varphi'}(T(f)) = false$.
  La trasformazione genera una nuova variabile a tale per cui $\varphi'(a) = false$ e tale variabile rende false tutta la valutazione $\Phi_{\varphi'}(\cdot)$.

5.9 E.9

Illustrare un qualche aspetto fondamentale della dimostrazione che ***CNF*** $\leq_P$ ***3CNF***.

Le formule **3CNF** sono formule in *conjunctive normal form* le cui clausole sono composte esattamente da 3 letterali. Ad esempio $[x][\overline{x}] \notin C$ mentre $[x, y, z], [k, \overline{z}, y] \in C$. La funzione di riduzione da **CNF** a **3CNF** si basa sull'idea di aggiungere variabili che però non intruducano vincolo alla soddifacibilità. Ci possono quindi essere 4 casi ed altrettante strategie di riduzione.Di seguito le varie strategie ed una intuizione della dimostrazione della funzione di riduzione.

1. (caso base).$F([l_1, l_2, l_3]) = [l_1, l_2, l_3]$. Se una clausola è già composta di soli letterali non serve effettuare alcuna trasformazione.

2. (caso base).$F([l_1]) = [l_1, y, z], [l_1, y, \overline{z}], [l_1, \overline{y}, z], [l_1, \overline{y}, \overline{z}]$. Essendoci un solo letterale vengono aggiunte 2 variabili e 4 clausole.Ogni clausola contiene una possibile combinazione dei valori delle due variabili.
   Risulta quindi completamente irrilevante ai fini della soddisfacibilità il valore assunto da x e y.Se$\varphi(l_1) = true$ qualsiasi combinazione di valori assunti da x e y (diciamo ad esempio $\varphi(x) = v_x$ e $\varphi(y) = v_y$) genererà clausole di tale forma:$[true, v_1, v_2]$, cioè clausole il cui valore di verità è $true$.

3. (caso base).$F([l_1, l_2]) = [l_1, l_2, y], [l_1, l_2, \overline{y}]$. Segue lo stesso discorso di prima, viene introdotta una variabile x che non ha alcun impatto sulla soddisfacibilità.

4. (caso induttivo, ci sono più di 3 letterali in una clausola).

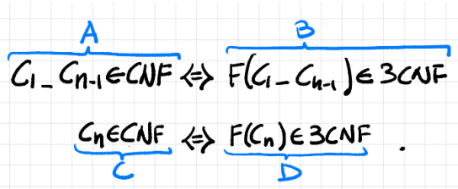
   $$F([l_1, l_2, l_3, ..., l_{k-2}, l_{k-1}, l_k] = [l_1, l_2, y], F([\overline{y}, l_3, ..., l_{k-2}, z]), [\overline{z}, l_{k-1}, l_k]$$

   In questo caso si divide la clausola in 3 sottogruppi. La clausola iniziale contiene i primi due letterali ed una nuova variabile, la clausola finale gli

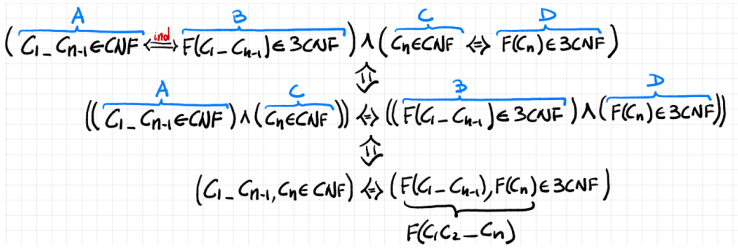
ultimi 2 letterali ed una nuova variabile.
Le due nuove varibili sono poi aggiunte negate alla clausola "centrale" su cui (avendo essa lunghezza > 2 e $<$ inf) viene applicata ricorsivamente la funzione di riduzione, avvalendosi della strategia 2 o della strategia 3 stessa. Le due variabili nuove non hanno impatto sulla soddisaficibilità della formula in quanto possiamo sfruttare la "libertà" che abbiamo di scegliere un valore di verità conveniente per y e z.
Prendiamo il caso peggiore: tutte i letterali tranne 1 sono falsi, solo uno è vero, chiamiamolo l_x.

- $l_x \in \{l_1, l_2\}$.Si possono scegliere y e z t.c.$\varphi'(\overline{y}) = \varphi'(\overline{z}) = true$.In tal modo la prima clausola è vera grazie a l_x e le altre due sono vere grazie a y e z
- $l_x \in \{l_3, ..., l_{k-2}\}$.Si possono scegliere y e z t.c.$\varphi'(y) = \varphi'(\overline{z}) = true$.In tal modo la prima clausola è vera grazie a y, la seconda grazie a l_x e la terza grazie a z
- $l_x \in \{l_{k-1}, l_k\}$.Si possono scegliere y e z t.c.$\varphi'(y) = \varphi'(z) = true$.In tal modo la prima clausola è vera grazie a y, la seconda grazie a z e la terza grazie ad l_x

5. (caso induttivo).$F(C_1, ..., C_n) = F(C_1), ..., F(C_n)$.Cioè la funzione di riduzione può essere apllicata a tutte le clausole di una formula senza modificarne la soddisfacibilità.



Dato che la formula $(A \Leftrightarrow B \wedge C \Leftrightarrow D) \Leftrightarrow ((A \wedge C) \Leftrightarrow (B \wedge D)$ è una tautologia, si può scrivere:



5.10 E.10

Nell'ambito della dimostrazione ***3CNF*** $\leq_P$ ***3COL****, definire i tratti fondamentali della riduzione* ***L*** *da formule a grafi e l'intuizione che ne guida la definizione.*

Lo scopo della funzione di riduzione $L : 3CNF \rightarrow 3COL$ è quello di generare un grafo partendo da una formula **3CNF**, tale grafico può essere colorato con 3 colori senza che due nodi adiacenti abbiano lo stesso colore se la formula di partenza è soddisfacibile, è incolorabile in caso contrario.

Definizione. *Sia* $G = (V, E)$*, grafo non orientato* $\exists c : Vertex \rightarrow \{col1, col2, col3\}$ *t.c.* $\forall u, v \in V \ e \ (u, v) \in E c(v) \neq c(u)$.

La funzione di riduzione L può essere introdotta e definita grazie a tre intuizioni da cui derivano altrettante restrizioni sul grafo.

1. Il primo passo consiste nel creare un grafo con una precisa struttura. A parte il nodo N ed i nodi F, T (perlaltro omissimibili) tutti gli altri nodi sono i letterali $l_1, ..., l_n$ così come compaiono nella formula **3CNF**, tutti collegati solamente al nodo N.

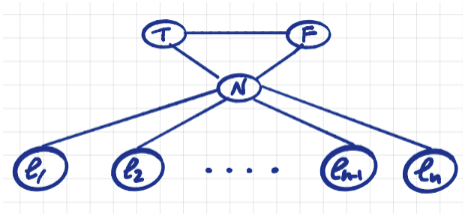


Figura 31: Grafo al passo 1

   Si nota che il grafo non risolve il problema da cui siamo partiti, contiene anzi una contraddizione, in quanto se nella formula dovessero ad esempio apparire due letterali $x, \hat{x}$ sarebbe possibile assegnargli lo stesso colore.

2. Quindi il secondo passo risolve il problema solevato dal primo, inserendo tra letterali"complementari" un arco.

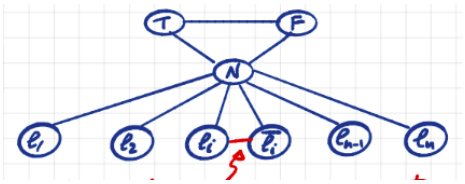


Figura 32: Grafo al passo 2

   Anche questo grafo no risolve completamente il problema, in quanto è possibile comunque colorare grafi derivanti da formule non soddisfacibili.

3. La rielaborazione finale del grafo prevende di aggiungere tante copie della rete in figura 33 quante sono le clausole della formula da cui si è partiti, collegando coerentemente i nodi a secondo di come i letterali compaiono nella formula

Ad esempio un grafo derivante dalla formula: $[x, a, b][x, \hat{a}, b][x, a, \hat{b}][x, \hat{a}, \hat{b}]$ può essere il seguente (il grafo e la colorazione non sono univoci).

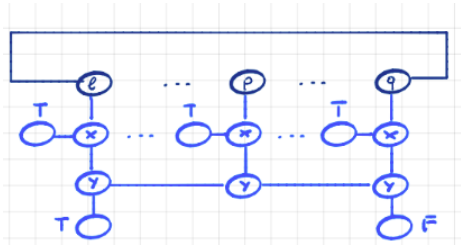


Figura 33: Grafo al passo 3

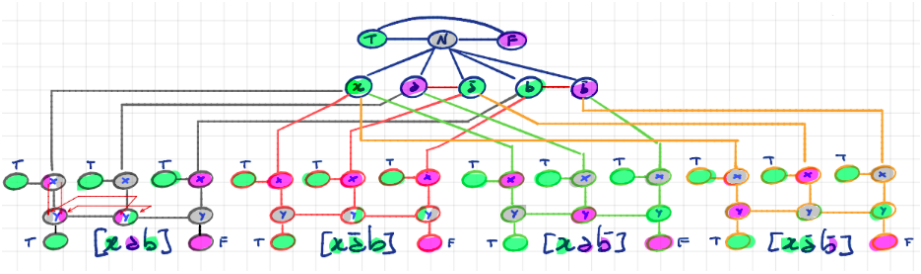


Figura 34: Esempio di grafo

Intuitivamente si può dire che questo metodo di costruzione del grafo è una funzione di riduzione in quanto se una singola clausola è soddisfcibile sicuramente uno dei 3 nodi che stanno di fianco al "nodo centrale y" hanno colore uguale, dato che almeno un letterale è vero (verde).
Se invece una clausola non è soddisfacibile tutti i letterali del sottografo generato al passo 3 hanno lo stesso colore (viola) ed è quindio impossibile assegnare un colore distinto ai 3 nodi y. Essendoci una clausola insoddisfacibile tutta la formula lo è.

5.11 E.11

Nell'ambito della dimostrazione ***3COL*** $\leq_P$ ***EXCO****, definire i tratti fondamentali della riduzione* E_x *da grafi ad insiemi e l'intuizione che ne guida la definizione.*

Dati U ed $S \subseteq \mathcal{P}(U)$ esiste un sottoinsieme $S' \subseteq S$ che è una copertura esatta di U cioè tra gli insiemi elementi di S compaiono tutti ed una sola volta gli elementi di U:

$$\forall z, y \in S' \, x \neq y \quad x \cap y = \emptyset \quad \bigcup_{x \in S'} X$$

La funzione di riduzione E_x da **3COL** a **EXCO** si basa sulla seguente intuzione:
Partendo dal grafo non colorato andiamo ad elencare tutte le possibili colorazioni di un nodo, aggiungendo una "sbavatura" in corrispondenza degli archi:
Per definizine di **3COL** bisogna evitare che le *sbavature* di due nodi con lo stesso colore si incontrino, sono invece accettabili incontri tra colori diversi.
A questi "incontri" di differenti colori è possibile assegnare dei nomi, il punto in

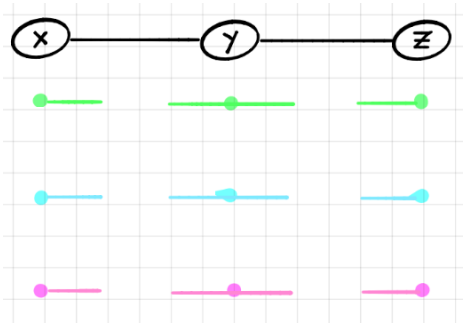
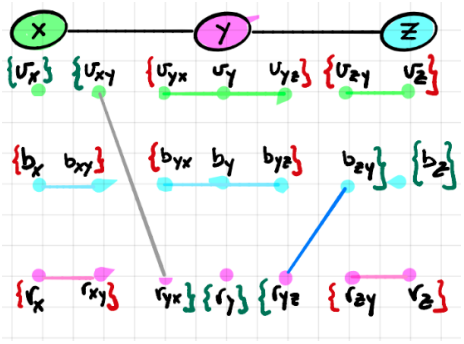


Figura 35: colorazioni di ogni nodo con sbavatura

cui il nodo x colorato verde si espande verso y (di colore non verde) può essere chiamato v_{xy} e viceversa in punto in cui il nodo y colorato ad esempio di rosso (ma andrebbe bene anche un altro colore che non sia verde, nell'esempio blu) si espande verso il nodo x si può chiamare r_{yx}. Da questo possiamo ricavare un insieme di "punti" che corrispondono a relazioni tra nodi o "incontri" e colore (r_x rosso x, b_y blu y, r_{yx} incorcio da y verso x colorato in rosso, etc ...). Questo insieme non è altro che l'insieme universo U, mentre una colorazione corrisponde ad S'.

Per generare un modello equivalente al grafico **3COL** bisogna fare attenzione ad inserire nello stesso sottoinsieme gli archi di "incontro" ($\{v_{xy}, r_{yx}\}$) in modo da avere in un altro sottoinsieme i singoletti che identificano il colore del nodo ($\{v_x\}$,$\{r_y\}$) ed in tal modo impedire a questi singoletti di entrare in altri sottoinsiemi. (se y è rosso voglio evitare che entri nel sottoinsieme $\{b_y x, b_y, b_y z\}$)



$$S' = \{\{v_x\}, \{v_{xy}, r_{yx}\}, \{r_y\}, \{r_{yz}, b_{zy}\}, \{b_z\}, \{b_x, b_{xy}\},$$
$$\{b_{yx}, b_y, b_{yz}\}, \{v_{yx}, v_y, v_{yz}\}, \{v_{zy}, v_z\}, \{r_x, r_{xy}\}\{v_{zy}, v_z\}\}$$

Questa intuizione comporta però un problema, in quanto l'insieme $S \subseteq \mathcal{P}(U)$ fornisce delle coperture di U che non corrispondono ad un grafo colorabile. (attenzione, fin'ora ci siamo concentrati su una specifica colorazione corrispondente ad S', ma l'insieme S ne può anche fornire altre).
La più notevole di queste coperture è la copertura in cui i singoletti v_x, b_y, r_z, etc..) non sono presenti, che sarebbe equivalente ad un grafo "non colorato" dal

momento che i singoletti indicano il colore di un vertice.

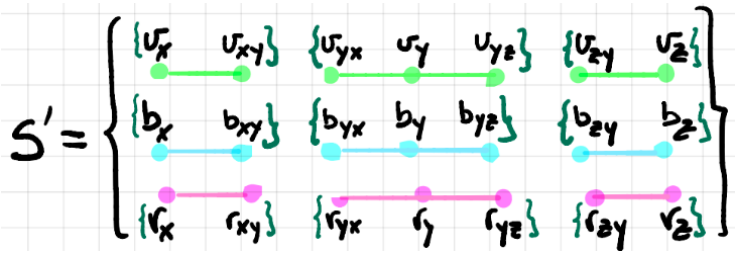


Figura 36: S' non è una colorazione

Possiamo quindi raffinare l'intuizione aggiungendo ad U tanti nodi quanti sono i vertici del grafo (nel nostro esempio aggiungiamo $\{x, y, z\}$) e utilizzandoli per decidere il colore di ogni nodo. Quindi da S rimuoviamo tutti i singoletti e aggiungiamo tutte le coppie variabile-singoletto (ad esempio $\{x, v_x\}, \{x, b_x\}, ..., \{y, r_y\}, ...$).

Formalmente otterremo, U^* e S^* "correzzioni" di U ed S:

$$U^* = \{x, y, z\} \cup U$$
$$S^* = (S\ \{\{v_x\}, \{b_x\}, \{r_x\}, \{v_y\}, \{b_y\}, \{r_y\}, \{v_z\}, \{b_z\}, \{r_z\}\})$$
$$\cup \{\{x, v_x\}, \{x, b_x\}, \{x, r_x\}, \{y, v_y\}, \{y, b_y\}, \{y, r_y\}, \{z, v_z\}, \{z, b_z\}, \{z, r_z\}\}$$

In tal modo è impossibile assegnare due colori allo stesso nodo (avremmo una ripetizione della variabile) ed è anche impossibile assegnare a due nodi adiacenti lo stesso colore (avremmo una intersezione di due insiemi o una non copertura).Ad esempio inserendo sia $\{x, v_x\}$ che $\{x, b_x\}$ avremmo x nell'intersezione ed avendo rimosso i singoletti è impossibile scegliere x_{colore} senza scegliere anche x.

5.12 E.12

Nell'ambito della dimostrazione ***EXCO*** $\leq_P$ ***Subsetsum****, definire i tratti fondamentali della riduzione* E_s *da insiemi a insiemi e l'intuizione che ne guida la definizione.*

Questa *funzione di riduzione* si basa sull'idea di "codificare" in binario la presenza o l'assenza degli elementi di S in S'.Quindi dato l'insieme degli elementi $U = s_1, s_2, ..., s_n$ andremo a trasformare S in un insieme composto da numeri binari di lunghezza n (equivalente a $|U|$).
Posto ad esempio $U = \{1, 2, 3\}$ un possibile S sarebbe $S = \{\{1, 2, 3\}, \{1, 3\}, \{\emptyset\}\}$ quindi $E_x(S) = \{\{1, 1, 1\}, \{1, 0, 1\}, \{0, 0, 0\}\}$.
La prima intuizione prosegue proponendo di sommare i tutti i numeri codificati in binario di S' e di confrontare il valore con $2^{|U|} - 1$, dato che la configurazione binaria di tale numero è 111...11 ripetuto $|U|$ volte la somma di tutti i numeri in S dovrebbe fornire $2^{|U|} - 1$ se e solo se sono presenti tutti gli elementi di U al massimo una volta.
tale intuizione risulta però errata in quanto non tiene conto del riporto, infatti:

$$\begin{array}{rccc} & 0 & 1 & 0 \\ & 1 & 1 & 0 \\ + & 0 & 1 & 0 \\ \hline & 1 & 1 & 1 \end{array}$$

Come si può notare il riporto ha "simulato" la presenza del valore in terza posizione. La soluzione è utilizzare una base che non permetta il verificarsi di tale situazione, quindi prima di effettuare la riduzione basta contare la massima occorrenza di ogni elemento in S, aggiungere una unità (così non si verificherà mai *carry*) ed utilizzare quella come base.
Formalmente:

Definizione. *Dati* $((U,S)S')$ *t.c.* $S \subseteq \mathcal{P}(U), S' \subseteq S$*:*

$$E_s : Set \times Set \times Set \to \mathcal{P}(\mathbb{N}) \times \mathbb{N}$$

$$(U,S,S') \to (\{\sum_{i=o}^{|u|-1} p^t x[i] | z \in S'\}, \frac{p^{|u|}-1}{p-1})$$

5.13 E.13

Definizione (SubsetSum)**.** *Siano dati un insieme numerico finito* X *ed un numero* S*. Determinare l'esistenza di un sottoinsieme* $Y \subset X$*, tale che la somma di tutti gli elementi di* Y *sia pari ad* S*.*

Vedendolo come problema decisionale può essere riscritto come:

$$((w_1,...,w_n),S) \in SubsetSum \text{ se } \exists x_1,...,x_n \in \{0,1\}^n \text{ t.c. } \sum_{1 \le k \le n} x_k w_k = S$$

SubsetSum è un caso specifico di ***KP***, in particolare è un problema **Knapsack** in cui i profitti sono uguali ai pesi.

$$((w_1,...,w_n),S) \in SubsetSum \Rightarrow ((w_1,...,w_n),(w_1,...,w_n),S) \in KP$$

Dimostrazione:

> Per ipotesi $\exists x_1^{\cdot},...,x_n^{\cdot}$ $tc \sum_{j=1}^n w_j x_j = S$ cioè esiste una scelta di valori $x_1,...,x_n$ che risolve il problema. ($x_1^{\cdot},...,x_n^{\cdot}$ è un istanza di $x_1,...,x_n$)
>
> Dobbiamo dimostrare che esiste una soluzione del rispettivo problema ***KP*** cioè che esista una istanza di scelte la cui somma dei pesi è minore di S ma allo stesso tempo è la massima scelta possibile (è più grande del peso di ogni altra possibile istanza di scelte):
>
> $$\exists z_1^{\cdot},...,z_n^{\cdot} \; tc \forall y_1,...,y_n \sum_{j=1}^n w_j z_j \ge \sum_{j=1}^n w_j y_j \text{ e } \sum_{j=1}^n w_j z_j \le S$$
>
> Prendendo come istanza $z^{\cdot}$ l' istanza $z^{\cdot}$ definita in precedenza abbiamo la sicurezza che, per ipotesi, $\sum_{j=1}^n w_j z_j \le S$
>
> La tesi diventa quindi
>
> $$\forall y_1,...,y_n \sum_{j=1}^n w_j x_j^{\cdot} \ge \sum_{j=1}^n w_j y_j$$

Supponiamo per assurdo che esista una scelta che fornisce un peso maggiore di quello generato dall'istanza $x^{\cdot}$:

$$\exists y_1^{\cdot}, ..., y_n^{\cdot} \sum_{j=1}^{n} w_j y_j^{\cdot} > \sum_{j=1}^{n} w_j x_j^{\cdot}$$

Allora $sum_{1 \leq j \leq n} w_j y_j^{\cdot} > \sum_{j=1}^{n} w_j x_j^{\cdot} = S$, ed essendo necessario identificare una sequenza di scelte $z_1, ..., z_n$ t.c. $\sum_{j=1}^{n} w_j z_j \leq S$ non possiamo accettare $y^{\cdot}$ come istanza risolutiva del problema dal momento che:

$$\sum_{j=1}^{n} w_j y_j^{\cdot} > S$$

5.14 E.14

Dimostrare che i problemi *KP* e *Bounded-KP* sono equivalenti.
Cioè $KP \Leftrightarrow BKP$.
Dimostrazione:

$KP \Rightarrow BKP$: ogni problema *KP* è un problema *BKP*.
La dimostrazione è immediata in quanto un generico problema $((P_1, ..., P_n), (W_1, ..., W_n), C) \in KP$ può essere riscritto come $((P_1, ..., P_n), (W_1, ..., W_n), (1, ..., 1), C) \in BKP$.

$KP \Leftarrow BKP$: ogni problema *BKP* è un problema *KP*.
per prima cosa introduciamo $B_i + 1$ nuove variabili, $\forall i \in B_1 ... B_n$.
Avremo quindi

$$x'_0, ..., x'_{b_i} \in \{0, 1\}$$
$$\vdots$$
$$x^n_0, ..., x^n_{b_i} \in \{0, 1\}$$

Queste variabili indicano "quanti" oggetti di tipo x^i sto prendendo, quindi

$$x^i_j = \begin{cases} 0 \; se \; x_i = 0 \\ 1 \; se \; x_i = j \end{cases}$$

Il problema è diventato un esempio di *BKP* di questo tipo:

$$((p_1, 2p_1, ..., b_1 p_1, ..., p_n, 2p_n, ..., b_n p_n), (w_1, 2w_1, ..., b_1 p_1, ..., w_n, 2w_n, ..., b_n w_n), C))$$
$$maximize \; \sum_{i=1}^{n} \sum_{j=0}^{b_i} p_j * j * x^i_j$$
$$subject \; to \; \sum_{i=1}^{n} \sum_{j=0}^{b_i} w_j * j * x^i_j$$
$$\text{con la condizione aggiuntiva } \forall i \sum_{j=0}^{b_i} x^i_j = 1$$

In modo che si possano prendere solo j oggetti fi tipo x^i DA FINIRE CON OSSERVAZIONE DEL PROF RIGUARDO AL FATTO DI DOVER DIMOSTRARE LA RISOLVIBILITà

6 Problemi computazionali

Nell'ambito della riduzione ***"Sotto-insiemi a Somma Identica"*** $\leq_P$ ***QUBO****, descrivere i passi fondamentali del processo che trasforma un'istanza del primo problema in una del secondo, sottolineando il ruolo giocato dall'elemento al quadrato finale.*

Nell'ambito della riduzione ***MAXCUT*** $\leq_P$ ***QUBO****, descrivere i passi fondamentali del processo che trasforma un'istanza del primo problema in una del secondo.*

Illustrare la tecnica che permette di trasformare tipici vincoli, che si possono presentare nella formulazione di problemi in ***PLI****, in modelli* ***QUBO****.*

Nell'ambito della riduzione ***MVC*** $\leq_P$ ***QUBO****, descrivere i tratti fondamentali del processo che trasforma un'istanza del primo problema in una del secondo, sottolineando il ruolo del lagrangiano scelto.*

Nell'ambito della riduzione ***Max2SAT*** $\leq_P$ ***QUBO****, descrivere i tratti fondamentali del processo che trasforma un'istanza del primo problema in una del secondo.*

Nell'ambito della riduzione di un generico problema in ***PLI*** *a* ***QUBO****, descrivere i tratti fondamentali del processo che trasforma un'istanza del primo problema in una del secondo, evidenziando i meccanismi di trasformazione dei vincoli in penalità.*

Nell'ambito della riduzione ***KP*** $\leq_P$ ***QUBO****, descrivere i tratti fondamentali della processo che trasforma un'istanza del primo problema in una del secondo.*