Algoritmi e Complessità

Michi Palazzo, Alessandro Soliman

Dipartimento di Informatica, Univerità degli Studi di Torino

Indice

Prologo, Algoritmica

Cos’è un algoritmo efficiente?

Definizione classica di efficienza:

Un algoritmo è efficiente se viene eseguito in tempo polinomiale su un computer **seriale**.

Tempo di esecuzione di algoritmi "efficienti":

$O(n)$, $O(n \log n)$, $O(n^3 \log^2 n)$

Tempo di esecuzione di algoritmi "inefficienti":

$O(2^n)$, $O(n!)$

Intrattabilità algoritmica / Intrattabilità

Un problema è chiamato **intrattabile** se, secondo le conoscenze attuali della teoria della complessità computazionale, non è risolvibile con un algoritmo in tempo polinomiale

I problemi intrattabili sono comuni. Dobbiamo discutere su come affrontarli quando li si incontra nella pratica.

Alcuni problemi **trattabili** (risolvibili con algoritmi noti in tempo polinomiale):

- Ricerca in un elenco non ordinato
- Ricerca in un elenco ordinato
- Ordinamento di un elenco
- Moltiplicazione di numeri interi

Alcuni problemi **intrattabili**:

- richiedono una quantità di output non polinomiale:
  - Torri di Hanoi.
  - Elenco di tutte le permutazioni di n numeri.
- richiedono quantità polinomiali di output:
  - Determina se c'è una strategia vincente per il Bianco in una partita a dama su una scacchiera *n x n*.

Intrattabilità algoritmica / Problemi intrattabili?

Esistono problemi per i quali **non conosciamo** algoritmi in tempo polinomiale, ma non siamo in grado di **dimostrarne l'intrattabilità**.

Molti problemi del mondo reale rientrano in questa categoria.

Intrattabilità algoritmica / Ottimizzazione

Alcuni modi per ottimizzare un algoritmo intrattabile:

- Eliminare le simmetrie e incorporando euristiche. Queste idee cercano di far funzionare bene l'algoritmo nella pratica, su istanze tipiche, pur riconoscendo che i casi esponenziali siano ancora possibili.
- Risolvere le versioni più semplici/limitate del problema, evitando la complessità esponenziale.
- Usare un algoritmo probabilistico in tempo polinomiale: cercando la risposta giusta con probabilità molto alta, rinunciando alla correttezza del programma.
- Per i problemi di ottimizzazione, utilizzare un algoritmo di approssimazione in tempo polinomiale, perdendo la garanzia di trovare la risposta migliore.

Brute-Force

Il concetto di **"visita *Brute-Force*"** si basa sull'approccio esaustivo di esplorare tutte le possibili soluzioni in uno spazio di ricerca.

Garantisce la completezza dell'esplorazione dello spazio delle possibili soluzioni.

Brute-Force / Casi d'uso

A differenza di altre strategie, **Brute-Force** è applicabile a un'ampia varietà di problemi.

Le spese per progettare un algoritmo più efficiente potrebbero non essere sempre giustificate se il Brute-Force risolvesse tali problemi in un tempo accettabile.

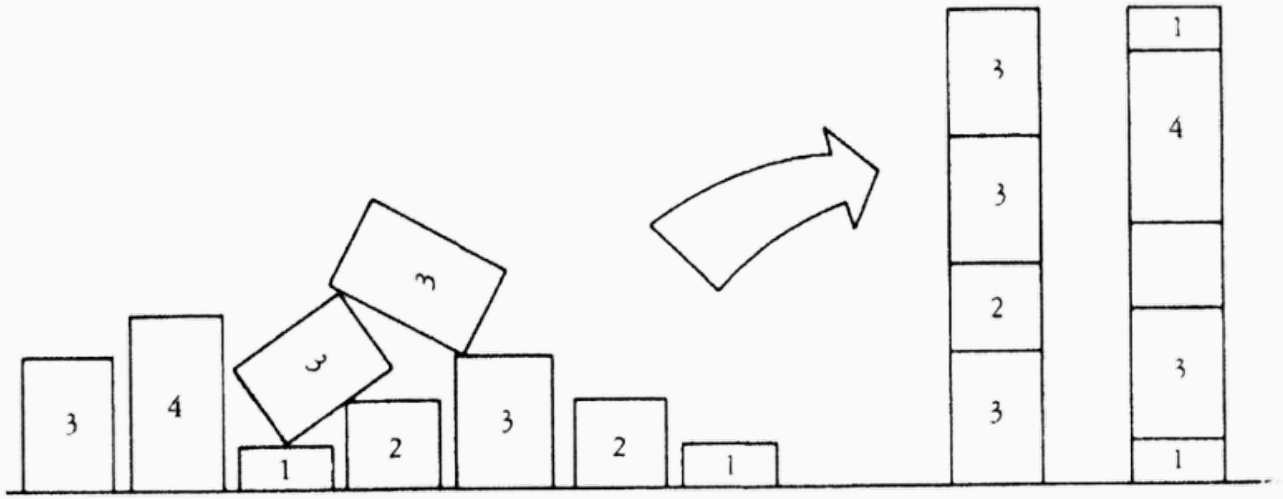
Inoltre, Brute-Force può servire a scopo teorico come metro di paragone con cui valutare alternative più efficienti.

Brute-Force / Codice

```
# candidates = insieme delle soluzioni possibili
# valid = predicato che verifica la validità
#         di una soluzione
def bruteforce(input):
  for solution in candidates:
    if valid(solution, input):
      return solution
  return None
```

Brute-Force / Esempio

Trovare un algoritmo per determinare se esistono due sottoinsiemi complementari di un insieme dato, la cui somma sia uguale.



Brute-Force / Spazio degli stati

Lo spazio degli stati è costituito da tutte le possibili soluzioni per un determinato input.

Ci sono diversi modi per esmprimere lo spazio degli stati:

$$\{[[1, 2, 3, 4, 5, 6], []], [[1, 2, 3, 4, 5], [6]], \ldots, [[3, 5, 6], [1, 2, 4]], \ldots, [[], [1, 2, 3, 4, 5, 6]]\}$$

$$\{[1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5], \ldots, [3, 5, 6], \ldots, []\}$$

$$\{[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1], \ldots, [1, 1, 0, 0, 1, 0], \ldots, [1, 1, 1, 1, 1, 1]\}$$

Brute-Force / candidates

```
def candidates(array):
    if len(array) == 0:
        yield []
        return
    for solution in candidates(array[1:]):
        yield candidate + [0]
        yield candidate + [1]
```

Brute-Force / valid

Se la soluzione è valida significa che è stata trovata una coppia di sottoinsiemi il cui totale è uguale; in caso contrario, la soluzione viene scartata e l'algoritmo continua a cercare altre soluzioni.

```
def valid(solution, array):
    sum = [0, 0]
    for (n, p) in zip(array, solution):
        sum[p] += n
    return sum[0] == sum[1]
```

Brute-Force / Soluzione

```
def partition(array):
    for solution in candidates(array):
        if valid(solution, array):
            return solution
    return None
```

Brute-Force / Conclusione

Vantaggi

- Facilità di implementazione
- Può essere usato per risolvere parti più piccole di problemi complessi

Svantaggi

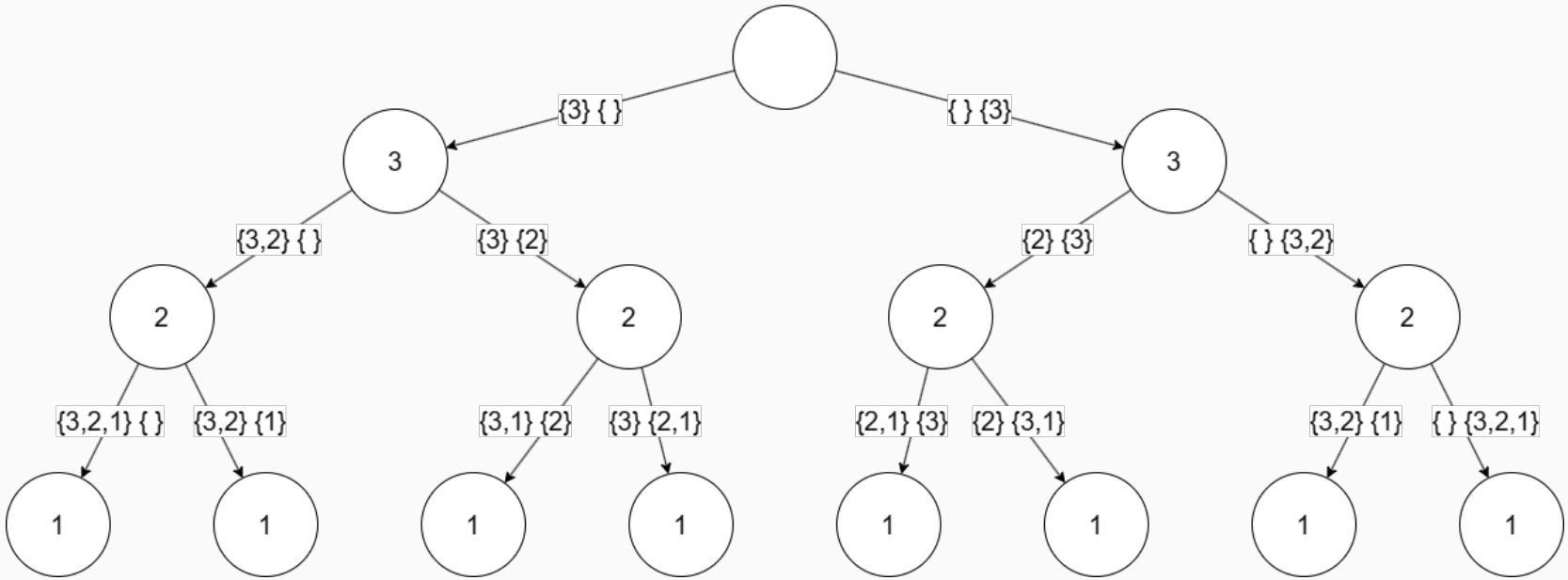
- Altamente inefficiente nel risolvere problemi complessi

Backtracking / Introduzione

Il *"Backtracking"* non è altro che una modifica dell'approccio di visita Brute-Force, in cui vengono scartati a priori insiemi di possibili soluzioni che non soddisfano determinati vincoli.

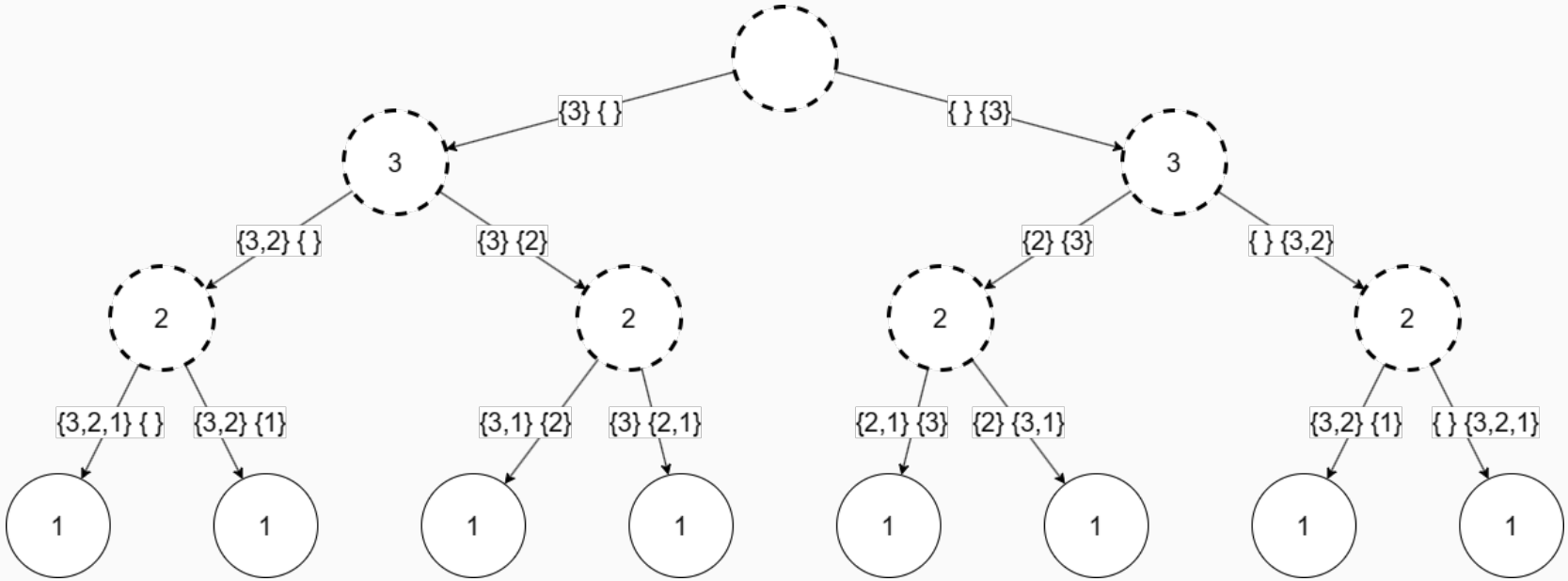
Backtracking / Albero degli stati

È possibile costruire un albero delle decisioni dove ogni nodo rappresenta una scelta di inclusione o esclusione di un elemento nell'insieme.



Backtracking / Soluzioni parziali

Le soluzioni parziali non possono essere delle soluzioni al problema ma ci aiutano a capire quali nodi possono essere scarati

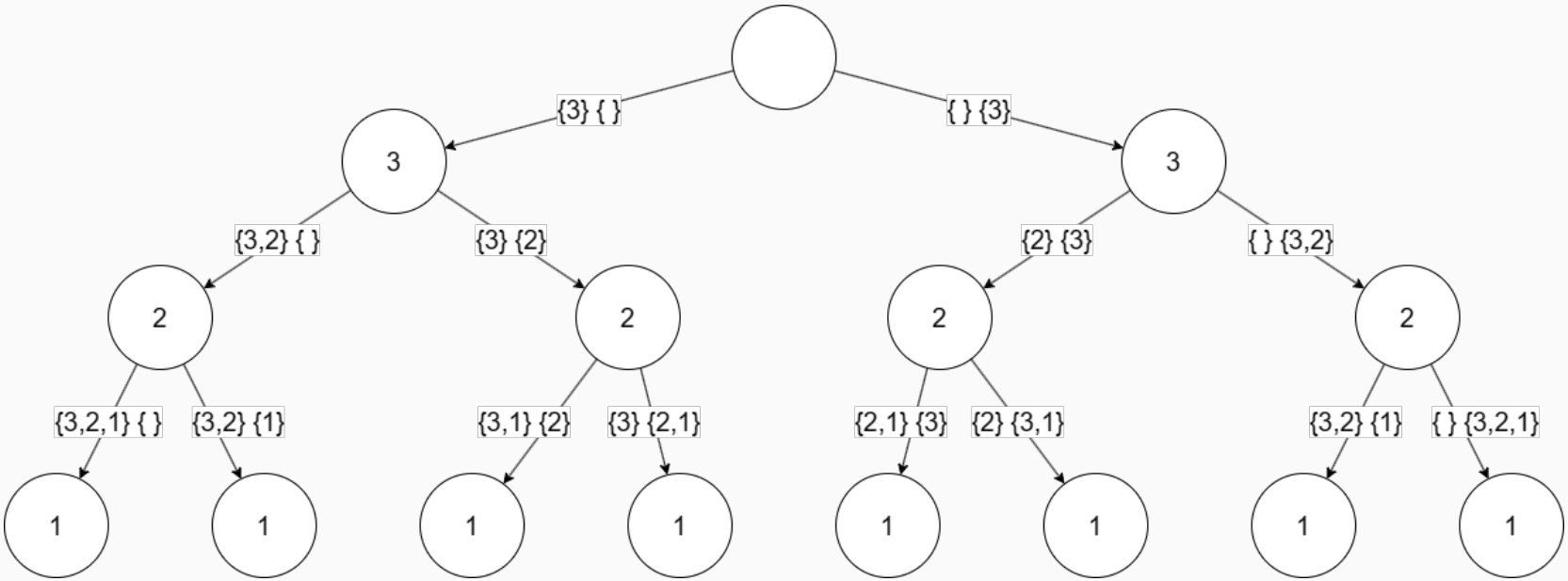


Backtracking / Pruning

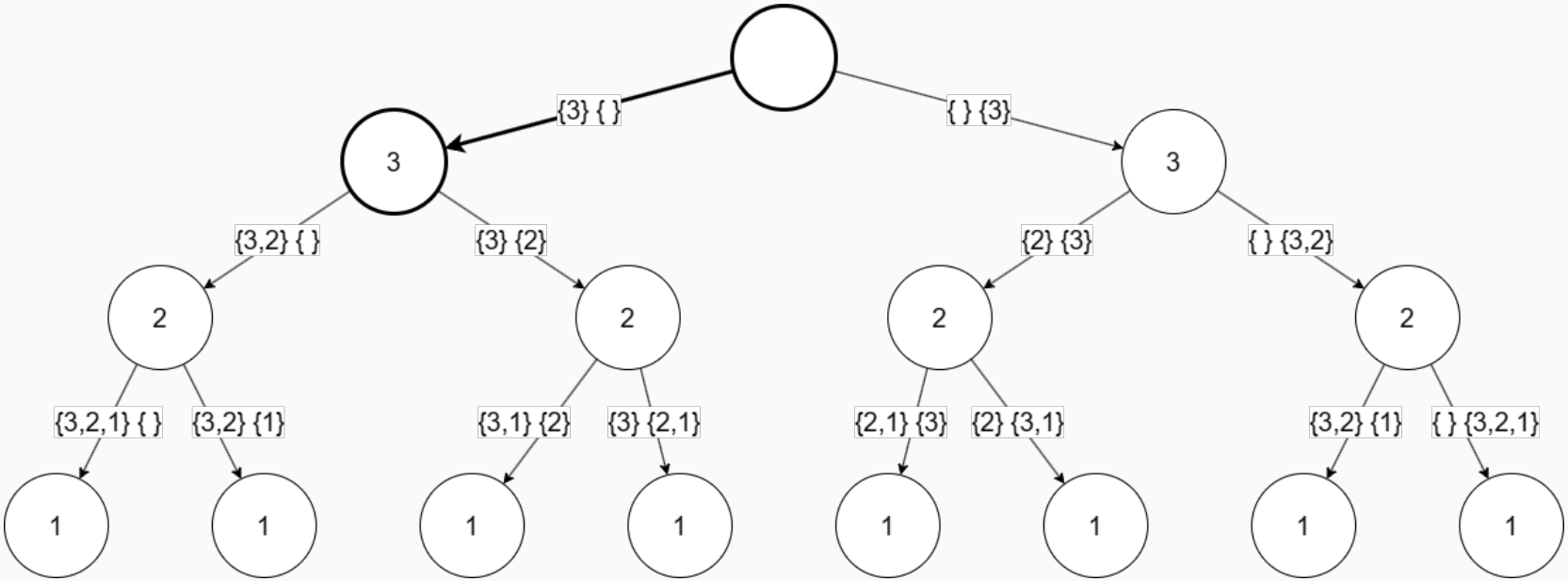
Il ***pruning*** consiste nel ridurre lo spazio di ricerca scartando soluzioni parziali che non possono portare a una soluzione valida o ottimale, migliorando così l'efficienza.

In questo caso, se la somma degli elementi attuali di un sottoinsieme supera la metà della somma totale degli elementi, non continua su quel ramo.

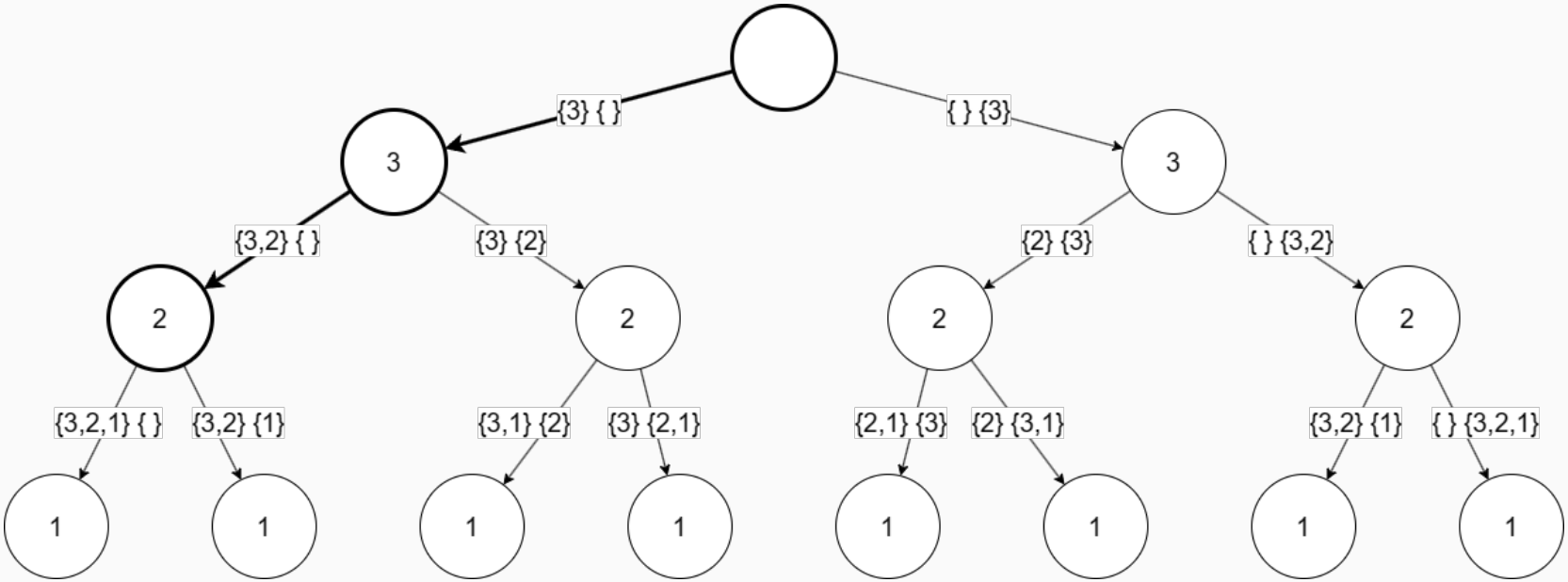
Backtracking / Pruning



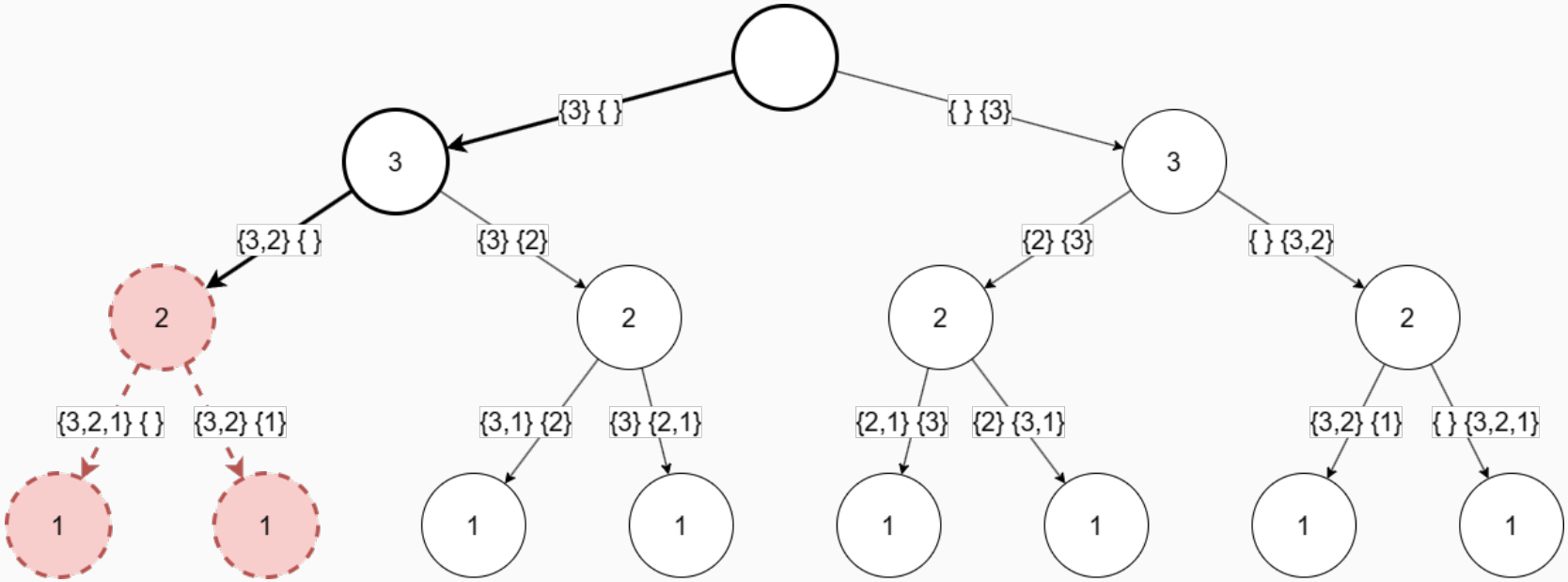
Backtracking / Pruning



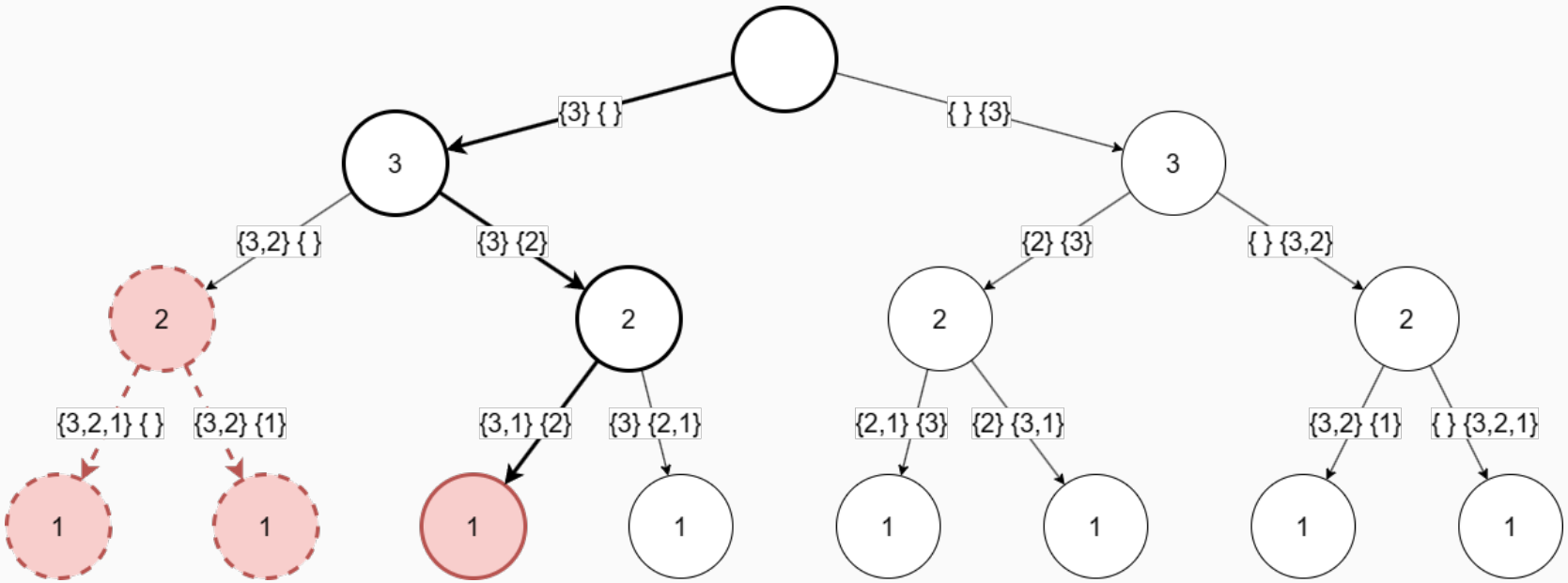
Backtracking / Pruning



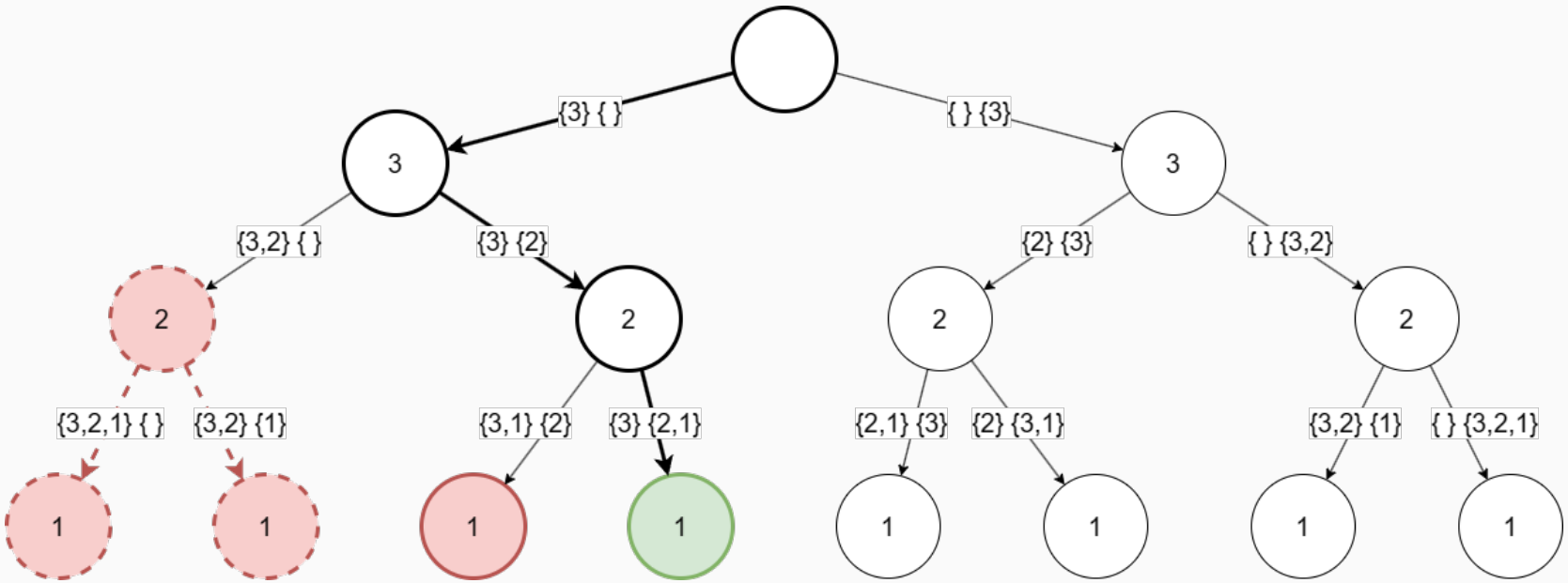
Backtracking / Pruning



Backtracking / Pruning



Backtracking / Pruning



Backtracking

```
def reject(branch, array):
    totals = [0, 0]
    for (n, p) in zip(array, branch):
        totals[p] += n
    remaining = sum(array[len(branch):])
    return totals[0] > totals[1] + remaining
        or totals[1] > totals[0] + remaining
```

Backtracking

```
def accept(solution, array):
    sum = [0, 0]
    for (n, p) in zip(array, solution):
        sum[p] += n
    return sum[0] == sum[1]
```

Backtracking

```
def expand(solution):
    yield solution + [0]
    yield solution + [1]
```

Backtracking

```
def complete(solution, array):
    return len(solution) == len(array)
```

Backtracking

```
def partition(array, solution=[]):
    if complete(solution, array):
        if accept(solution, array):
            return solution
        else:
            return None
    elif reject(solution, array):
        return None
    else:
        for branch in expand(solution):
            candidate = partition(array, branch)
            if candidate != None:
                return candidate
        return None
```

Backtracking / Conclusione

Vantaggi

- Risulta efficace per i problemi di soddisfacimento di vincoli

Svantaggi

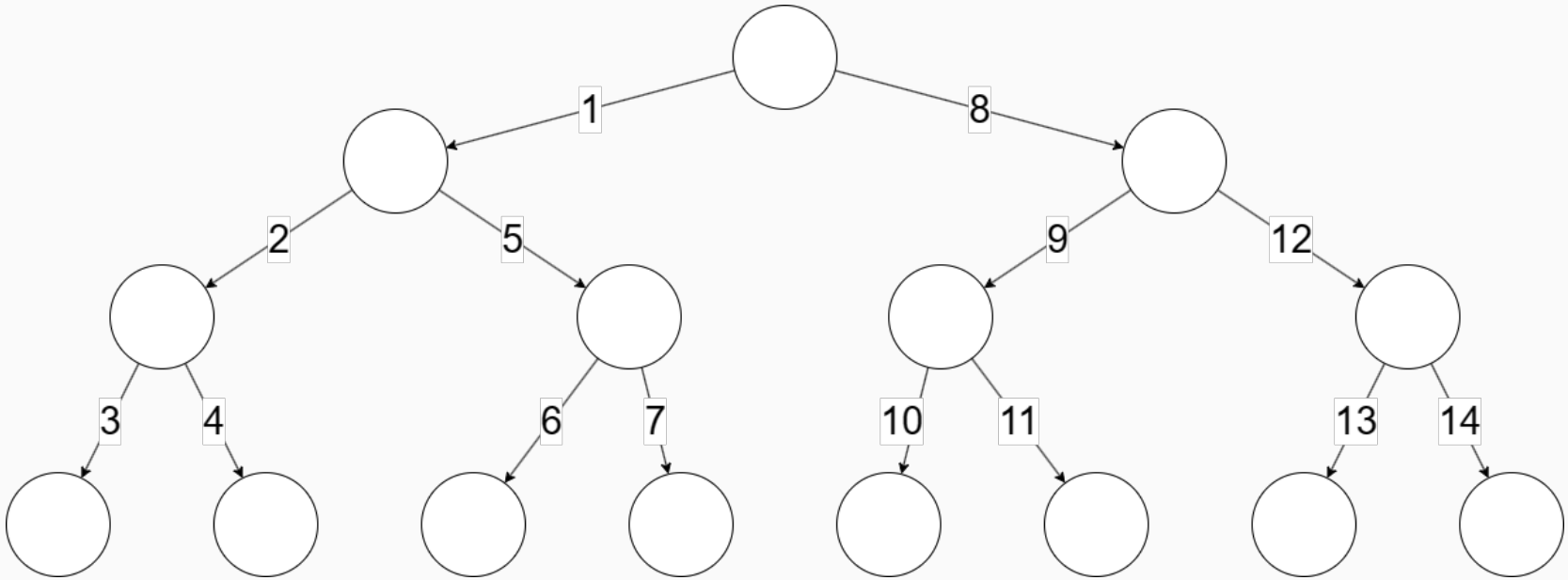
- Richiede molta memoria per la risoluzione di problemi di grandi dimensioni
- Inefficiente nel risolvere problemi complessi

Least-cost

Per superare le limitazioni imposte dal Backtracking, è possibile utilizzare criteri di generazione e visita dello spazio degli stati più efficienti. Tra questi, il criterio ***"Least-cost"*** si concentra sulla generazione e visita degli stati in modo tale da minimizzare il costo computazionale e migliorare le prestazioni complessive dell'algoritmo.

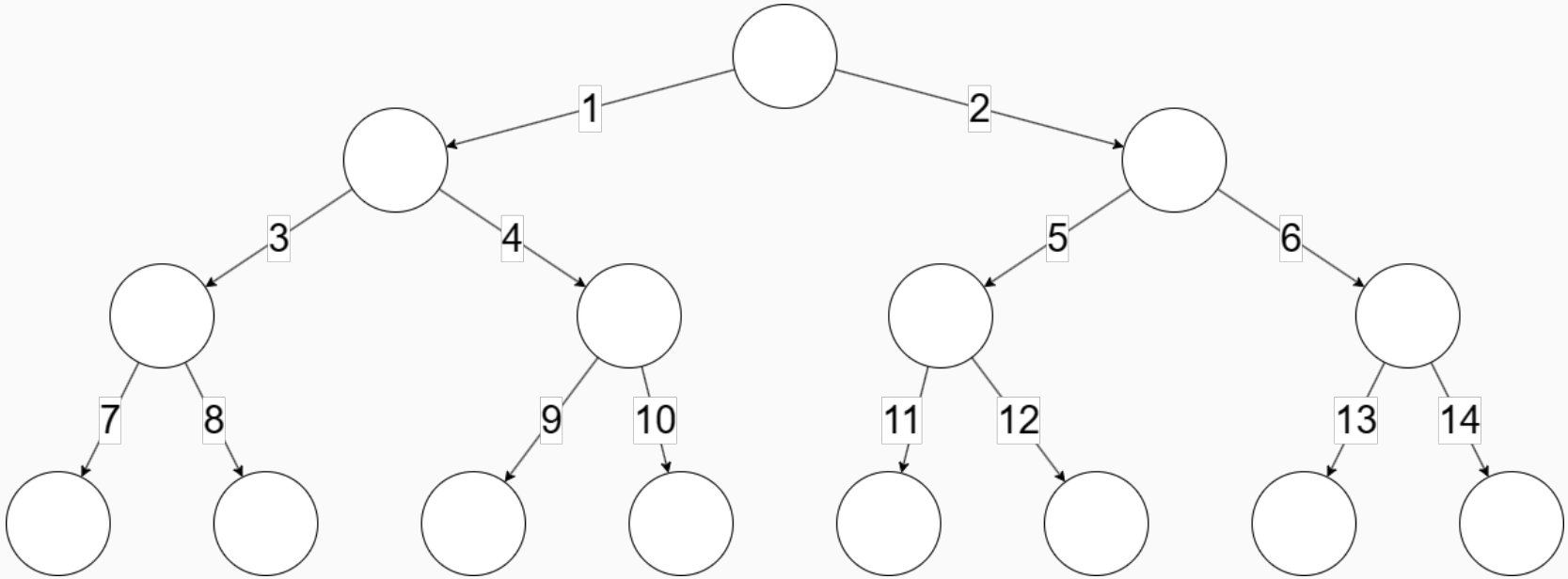
Least-cost

La *"DFS"* è una tecnica di ricerca che esplora un ramo dell'albero di stati fino alla sua profondità massima prima di retrocedere ed esplorare altri rami. Utilizza uno stack per tenere traccia dei nodi visitati e dei percorsi attuali.



Least-cost

La *"BFS"* è una strategia di esplorazione che visita tutti i nodi di un livello prima di passare al livello successivo. Utilizza una coda FIFO per mantenere traccia dei nodi da esplorare.



Least-cost

La *"Least Cost Search"* (ricerca a costo minimo) è una tecnica di esplorazione che visita prima i nodi con il costo cumulativo più basso. Utilizza una coda di priorità (spesso implementata come una heap) per mantenere i nodi da esplorare, ordinati per costo cumulativo crescente.

Backtracking / Visita non ricorsiva

```
def partition(array):
    stack = []
    push(stack, [])
    while len(stack) > 0:
        solution = pop(stack)
        if complete(solution, array):
            if accept(solution, array):
                return solution
        elif not reject(solution, array):
            for branch in expand(solution):
                push(stack, branch)
    return None
```

Least-cost

```
def cost(solution, prev_remaining):
    if solution[-1] == 0:
        n = array[len(solution) - 1]
        return abs(prev_remaining - n)
    return prev_remaining
```

Least-cost

```
def partition(array):
    heap = []
    push(heap, (sum(array) // 2, []))
    while len(heap) != 0:
        remaining, solution = pop(heap)
        if complete(solution, array):
            if accept(solution, array):
                return solution
        elif not reject(solution, array):
            for branch in expand(solution):
                push(heap, (cost(branch, remaining)
    return None
```

Least-cost

L’approccio ”least cost” può offrire vantaggi significativi per specifici tipi di input e problemi ben strutturati con euristiche efficaci. Tuttavia, può anche risultare inefficace o addirittura peggiorare le prestazioni in casi dove le euristiche sono scarse o i costi sono uniformi.

Least-cost

Vantaggi

- Migliore efficienza per input specifici

Svantaggi

- Implementazione più complessa

KP

Il *"problema dello zaino"* è un classico problema di ottimizzazione combinatoria che richiede di trovare la combinazione ottimale di oggetti da inserire in uno zaino, in modo da massimizzare il valore totale degli oggetti, rispettando però il vincolo del peso massimo che lo zaino può sopportare.

KP

Input:

- n: numero di oggetti
- $w_1, \ldots, w_n$: pesi degli oggetti
- $p_1, \ldots, p_n$: profitti degli oggetti
- C: capacità massima dello zaino

Variabili Decisionali:

- $x_k \in \{0, 1\}$ (inclusione oggetto k)

Formulazione del Problema

$$\text{Massimizzare:} \quad \sum_{k=1}^{n} p_k x_k$$

$$\text{Soggetto a:} \quad \sum_{k=1}^{n} w_k x_k \leq C$$

$$x_k \in \{0, 1\}, \quad k = 1, \ldots, n$$

KP

KP cattura l'essenza del concetto intuitivo di un processo decisionale di cui valutare l'efficacia.

Formalizza l'obiettivo di un processo decisionale tramite un insieme di vincoli su combinazioni lineari di valori.

Applicazioni Reali del KP

- **Investimento Finanziario:** Allocazione di risorse
- **Gestione di materiale:** minimizzare gli sprechi e massimizzare il profitto
- **Spedizioni:** massimizzare l'uso dello spazio di carico
- **Valutazione:** valutare test e ottimizzare scelte in base a criteri predefiniti

KP / problemi riconducibili e varianti

- Subsetsum
- **Bounded KP**: caratterizzato da variabili intere che però possono assumere valori nel segmento $[0, ..., n] \in \mathbb{N}$
- **Multi-dimensional KP**: in cui si hanno più di un limite superiore di “capacità” da non superare
- **Multiple KP**: è basato sull’idea che ci sono più zaini da riempire
- **Multiple-choice KP**: basato sull’idea che a fronte della disponibilità di un solo zaino, abbiamo diversi tipi di item e, per ogni tipo, abbiamo più item di diverso peso.

KP / algoritmi greedy

Un algoritmo *"greedy"* è un metodo di risoluzione di problemi che, a ogni passaggio, fa una scelta che sembra la migliore al momento, senza preoccuparsi delle conseguenze a lungo termine.

Questa scelta viene fatta seguendo una regola di selezione specifica, che dipende dal problema che si sta affrontando.

KP / algoritmi greedy

Algoritmi che risolvono il KP scegliendo iterativamente l'elemento con il miglior rapporto profitto/peso.

Esempi di Algoritmi Greedy:

- Greedy
- Greedy-Split
- Extended-Greeedy
- Linear relaxation

KP / greedy

```python
def kp_greedy(array, capacity):
    sorted_array = sorted(
        array,
        key=lambda item: item.profit / item.weight,
        reverse=True
    )
    profit = 0
    weight = 0
    for item in sorted_array:
        if weight + item.weight <= capacity:
            profit += item.profit
            weight += item.weight
    return profit
```

KP / extended greedy

```
def kp_extended_greedy(array, capacity):
    max_profit = kp_greedy(array, capacity)
    for item in array:
        if item.weight <= capacity and
           item.profit > max_profit:
            max_profit = item.profit
    return max_profit
```

Rilassamento Lineare LKP

- **Problema:**

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

$0, 1$ rappresenta l'intervallo dei numeri reali tra 0 e 1.

Algoritmo Greedy-LKP

- Ordina gli elementi per efficienza ($\frac{p_i}{w_i}$).
- Inserisce interamente gli elementi finché possibile.
- All'elemento di indice *split*, aggiunge la frazione:

$$\frac{C - \hat{w}}{w_{\text{split}}}$$

- Il profitto totale è:

$$z_{LP} = \hat{p} + \left(\frac{C - \hat{w}}{w_{\text{split}}}\right) p_{\text{split}}$$

- Il vettore risposta è:

$$x_{LP} = \left(1, \ldots, 1, \frac{C - \hat{w}}{w_{\text{split}}}, 0, \ldots, 0\right)$$

KP / linear relexation

```
def kp_linear(array, capacity):
    sorted_array = sorted(
        array,
        key=lambda item: item.profit / item.weight,
        reverse=True
    )
    profit = 0
    weight = 0
    for item in sorted_array:
        if weight + item.weight <= capacity:
            profit += item.profit
            weight += item.weight
        else:
            profit += (capacity - weight) *
                      item.profit / item.weight
            break
    return profit
```

Ottimalità Greedy-LKP

- La *greedy optimality property* garantisce che la scelta migliore locale assicura la soluzione globale ottimale.
- LKP massimizza il profitto considerando frazioni di oggetti.

Misura la **qualità della risposta**. Questo criterio di valutazione si basa sul rapporto tra il profitto massimo ottenuto dall'algoritmo applicato a una specifica istanza del problema e il profitto ottimale per quella stessa istanza. Un algoritmo greedy che fornisce una buona approssimazione avrà questo rapporto molto vicino a 1.

Un algoritmo greedy G è una **k-approximation**, con $0 \leq k \leq 1$, se per ogni istanza I del problema, il rapporto tra il profitto $z_G(I)$ garantito dall'algoritmo e il profitto ottimale $z_P^*(I)$ è almeno k. Questo può essere espresso come:

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

Una k-approximation è detta **tight** se esiste un'istanza T del problema per la quale il rapporto è esattamente k, cioè:

$$\frac{z_G(T)}{z_P^*(T)} = k.$$

L'algoritmo **Ext-Greedy** per il problema dello zaino fornisce una **1/2-approximation**. Questo significa che, per ogni istanza del problema, il profitto ottenuto dall'algoritmo è almeno la metà del profitto ottimale.

$$\frac{z_{Ext-Greedy}(I)}{z_P^*(I)} \geq \frac{1}{2}.$$

L'algoritmo mantiene una performance di qualità ragionevole rispetto alla soluzione ottimale.

KP / Tightness

La proprietà di 1/2-approximation di Ext-Greedy è **tight**. Esiste infatti un'istanza specifica del problema per la quale il rapporto tra il profitto ottenuto dall'algoritmo e il profitto ottimale è esattamente 1/2, non è quindi possibile migliorare l'approssimazione oltre questa soglia utilizzando l'algoritmo Ext-Greedy.

KP / Branch&Bound

"Branch&Bound" è una tecnica completa che garantisce di trovare una soluzione ottimale.

Inizia con una soluzione fornita da un algoritmo greedy, che fornisce un primo lower bound del profitto. Da qui, l'albero di ricerca è esplorato espandendo i nodi, calcolando i limiti superiori e inferiori per ogni sottoproblema. Se il limite superiore di un sottoproblema è inferiore alla soluzione corrente, quel ramo dell'albero è potato.

Il calcolo dell'**upper bound** si basa su una funzione di costo che prevede il profitto massimo possibile ancora ottenibile da un sottoproblema. Questa previsione si ottiene sommando il profitto corrente al massimo profitto che potrebbe essere ottenuto aggiungendo ulteriori elementi nello zaino. Questo valore è spesso determinato utilizzando un approccio greedy applicato ai rimanenti elementi.

KP / Branch&Bound

```
def reject(branch, array, capacity, max_profit):
    weight = 0
    profit = 0
    for (x, p) in zip(array, branch):
        weight += x.weight * p
        profit += x.profit * p
    return weight > capacity or profit + kp_linear(
        array[len(branch):],
        capacity - weight
    ) < max_profit
```

KP / Branch&Bound

```
def kp(array, capacity):
    max_profit = kp_extended_greedy(array, capacity)
    heap = []
    push(heap, (0, []))
    while len(heap) > 0:
        profit, solution = pop(heap)
        if reject(solution, array, capacity, max_profit):
            continue
        if complete(solution, array):
            if profit > max_profit:
                max_profit = profit
            continue
        for branch in expand(solution):
            push(heap, (cost(branch, array, profit), branch))
    return max_profit
```

KP / DP

La *"Programmazione Dinamica"* (DP) è una tecnica di ottimizzazione multi-fase introdotta da Bellman negli anni '50.

Si basa su un metodo equazionale ricorsivo e un principio di ottimalità.

DP si applica risolvendo problemi complessi suddividendoli in sottoproblemi più semplici e risolvendo ogni sottoproblema una sola volta.

Equazione Generale di DP:

$$f(\overline{g}) = \max_{\overline{x}} [H(\overline{g}, \overline{x}, f(T(\overline{g}, \overline{x})))] \tag{1}$$

- x: Vettore delle porzioni di risorse da cercare per massimizzare H.
- g: Vettore di parametri, es. ricavi per ogni attività.
- $T(g, x)$: Funzione predecessore che riduce il problema di ottimizzazione.

Il principio di ottimalità per DP afferma che una soluzione ottimale globale può essere costruita da soluzioni ottimali di sottoproblemi.

Questo principio generalizza il ***"Greedy Optimality Principle"***:

$$f_N(x) = \max_{0 \leq x_N \leq x} [g_N(x_N) + f_{N-1}(x - x_N)] \tag{2}$$

Ogni decisione locale (allocazione di risorsa) contribuisce alla soluzione globale ottimale.

Passi del Processo Ricorsivo:

$$\begin{aligned} f_N(x) &= g_N(x_N) + f_{N-1}(x - x_N) \\ f_{N-1}(x - x_N) &= g_{N-1}(x_{N-1}) + f_{N-2}((x - x_N) - x_{N-1}) \\ &\vdots \\ f_k(x) &= g_k(x_k) + f_{k-1}(x - x_k) \end{aligned} \tag{3}$$

Inizia con il caso base: $f_1(x_1) = g_1(x_1)$.

Ogni passo somma il ricavo corrente al ricavo del sottoproblema rimanente.

KP / DP

```
def kp(array, capacity):
    n = len(array)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if array[i - 1].weight <= w:
                dp[i][w] = max(
                    dp[i - 1][w],
                    dp[i - 1][w - array[i - 1].weight]
                        + array[i - 1].profit
                )
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][capacity]
```

KP / DP / Esempio

- oggetto 1: peso = 1, valore = 6
- oggetto 2: peso = 2, valore = 10
- oggetto 3: peso = 3, valore = 12
- capacità massima: 5

i \ w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3	0	6	10	16	18	22

La complessità temporale di DP per KP è $O(n \cdot C)$, dove n è il numero di elementi e C è la capacità dello zaino.

Pseudo-polinomiale: Polinomiale rispetto al numero di elementi, esponenziale rispetto al valore numerico di C.

$$O(n \cdot 2^{1+\lceil \log_2 C \rceil}) \tag{4}$$

KP ha un ***"Polynomial Time Approximation Scheme"*** (PTAS), rendendolo uno dei problemi più semplici tra quelli intrattabili.

La tabella delle decisioni può essere evitata identificando i punti chiave:

- Confrontare i profitti con e senza l'inserimento di un elemento.
- Identificare le situazioni in cui il profitto non cambia.

Riduzione dello spazio: Conservare solo i profitti, non le decisioni.

Riduzioni, Complessità computazionale

Riduzione polinomiale

La riduzione polinomiale è una tecnica che permette di trasformare un problema computazionale in un altro in tempo polinomiale.

Riduzione polinomiale / Motivazioni

- **Classificazione della Complessità**: Identificare problemi che appartengono alla stessa classe di complessità.
- **Trasferibilità delle Soluzioni**: Se un problema A può essere ridotto a un problema B e B è risolvibile in tempo polinomiale, allora A è risolvibile in tempo polinomiale.
- **Dimostrazione di NP-completezza**: Utilizzata per dimostrare che un problema è NP-completo mostrando che ogni problema in NP può essere ridotto ad esso in tempo polinomiale.

Riduzione polinomiale / Vantaggi

- **Semplificazione delle Prove**: Facilita la dimostrazione che nuovi problemi sono NP-completi riducendoli a problemi già noti come NP-completi.
- **Unificazione della Complessità**: Fornisce un quadro unificato per studiare la complessità dei problemi mostrando che molti problemi apparentemente diversi hanno la stessa difficoltà computazionale.
- **Ottimizzazione degli Algoritmi**: Permette di applicare tecniche e algoritmi sviluppati per un problema ad altri problemi attraverso riduzioni.

Formula proposizionale (pf)

- $\{x_1, x_2, \ldots, x_n\} \in pf$
- $f \in pf \Longrightarrow \bar{f} \in pf$
- $f_1 \in pf \wedge f_2 \in pf \Longrightarrow f_1 \wedge f_2 \in pf$
- $f_1 \in pf \wedge f_2 \in pf \Longrightarrow f_1 \vee f_2 \in pf$

Data $f \in pf$ esiste un'assegnazione di valori di verità φ alle variabili in $FV(f)$ che soddisfa f, cioè tale che $\Phi_\varphi(f) = \text{true}$?

NNF

Formula proposizionale in NNF (nnf)

- $\{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\} \in nnf$
- $f_1 \in nnf \land f_2 \in nnf \Longrightarrow f_1 \land f_2 \in nnf$
- $f_1 \in nnf \land f_2 \in nnf \Longrightarrow f_1 \lor f_2 \in nnf$

Data $f \in nnf$ esiste un'assegnazione di valori di verità φ alle variabili in $FV(f)$ che soddisfa f, cioè tale che $\Phi_\varphi(f) = \text{true}$?

SAT $\leq_P$ NNF

Esiste l'algoritmo M : pf $\rightarrow$ nnf che trasforma l'input di **SAT** in un equivalente input di **NNF**

$$M(x) = x \qquad \forall x \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$$

$$M(f_1 \wedge f_2) = M(f_1) \wedge M(f_2) \qquad \forall f_1, f_2 \in pf$$

$$M(f_1 \vee f_2) = M(f_1) \vee M(f_2) \qquad \forall f_1, f_2 \in pf$$

$$M(\overline{f_1 \wedge f_2}) = M(\overline{f_1}) \vee M(\overline{f_2}) \qquad \forall f_1, f_2 \in pf$$

$$M(\overline{f_1 \vee f_2}) = M(\overline{f_1}) \wedge M(\overline{f_2}) \qquad \forall f_1, f_2 \in pf$$

SAT $\leq_P$ NNF / Dimostrazione

Sia $f \in pf$

$$
\begin{aligned}
&\exists \varphi : FV(f) \to \{\text{true}, \text{false}\} && \Phi_\varphi(f) = \text{true} && \iff \\
&\exists \varphi : FV(M(f)) \to \{\text{true}, \text{false}\} && \Phi_\varphi(f) = \text{true} && \iff \\
&\exists \varphi : FV(M(f)) \to \{\text{true}, \text{false}\} && M(\Phi_\varphi(f)) = M(\text{true}) && \iff \\
&\exists \varphi : FV(M(f)) \to \{\text{true}, \text{false}\} && \Phi_\varphi(M(f)) = \text{true}
\end{aligned}
$$

CNF

Formula proposizionale in CNF (cnf)

- $\{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\} \in C$
- $f \in C \wedge x \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\} \Longrightarrow f \vee x \in C$
- $c \in C \Longrightarrow c \in \mathit{cnf}$
- $f \in \mathit{cnf} \wedge c \in C \Longrightarrow f \wedge c \in \mathit{cnf}$

Data $f \in \mathit{cnf}$ esiste un'assegnazione di valori di verità φ alle variabili in $\mathit{FV}(f)$ che soddisfa f, cioè tale che $\Phi_\varphi(f) = \text{true}$?

NNF $\leq_P$ CNF

Esiste l'algoritmo di Tseytin T : nnf $\rightarrow$ cnf che trasforma l'input di **NNF** in un equivalente input di **CNF**

$$V(f) = x_{\text{new}} \qquad \forall f \in nnf$$

$$C(v, x) = (v \Leftrightarrow x) \qquad \forall x \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$$
$$= (\overline{v} \vee x) \wedge (\overline{x} \vee v)$$

$$C(v, f_1 \vee f_2) = (v \Leftrightarrow V(f_1) \vee V(f_2)) \wedge C(V(f_1), f_1) \wedge C(V(f_2), f_2) \qquad \forall f_1, f_2 \in nnf$$
$$= (\overline{v} \vee V(f_1) \vee V(f_2)) \wedge (\overline{V(f_1)} \vee v) \wedge (\overline{V(f_2)} \vee v) \wedge C(V(f_1), f_1) \wedge C(V(f_2), f_2)$$

$$C(v, f_1 \wedge f_2) = (v \Leftrightarrow V(f_1) \wedge V(f_2)) \wedge C(V(f_1), f_1) \wedge C(V(f_2), f_2) \qquad \forall f_1, f_2 \in nnf$$
$$= (v \vee \overline{V(f_1)} \vee \overline{V(f_2)}) \wedge (V(f_1) \vee \overline{v}) \wedge (V(f_2) \vee \overline{v}) \wedge C(V(f_1), f_1) \wedge C(V(f_2), f_2)$$

$$T(f) = V(f) \wedge C(V(f), f) \qquad \forall f \in nnf$$

NNF $\leq_P$ CNF / Dimostrazione

Sia $f \in nnf$

$\exists \varphi : FV(f) \to \{\text{true}, \text{false}\}$ $\qquad \Phi_\varphi(f) = \text{true} \quad \iff$

per induzione simultanea (**Lemma T**)

$\exists \varphi' : FV(T(f)) \to \{\text{true}, \text{false}\}$ $\qquad \Phi_{\varphi'}(T(f)) = \text{true}$

3CNF

Formula proposizionale in 3CNF (3cnf)

- $x, y, z \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\} \Longrightarrow x \vee y \vee z \in C$
- $c \in C \Longrightarrow c \in 3cnf$
- $f \in 3cnf \wedge c \in C \Longrightarrow f \wedge c \in 3cnf$

Data $f \in 3cnf$ esiste un'assegnazione di valori di verità φ alle variabili in $FV(f)$ che soddisfa f, cioè tale che $\Phi_\varphi(f) = \text{true}$?

CNF $\leq_P$ 3CNF

Esiste F : cnf $\rightarrow$ 3cnf che trasforma l'input di **CNF** in un equivalente input di **3CNF**

$$F(x_1) = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \qquad \forall x_1 \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$$

$$F(x_1 \vee x_2) = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \qquad \forall x_1, x_2 \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$$

$$F(x_1 \vee x_2 \vee x_3) = x_1 \vee x_2 \vee x_3 \qquad \forall x_1, x_2, x_3 \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$$

$$F(f \vee x_1 \vee x_2 \vee x_3) = (x_1 \vee x_2 \vee y) \wedge F(\overline{y} \vee f) \qquad \forall f \in C, x_1, x_2, x_3 \in \{x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\}$$

$$F(f \wedge c) = F(f) \wedge F(c) \qquad \forall f \in 3cnf, c \in C$$

CNF $\leq_P$ 3CNF / Dimostrazione

Sia $f \in cnf$

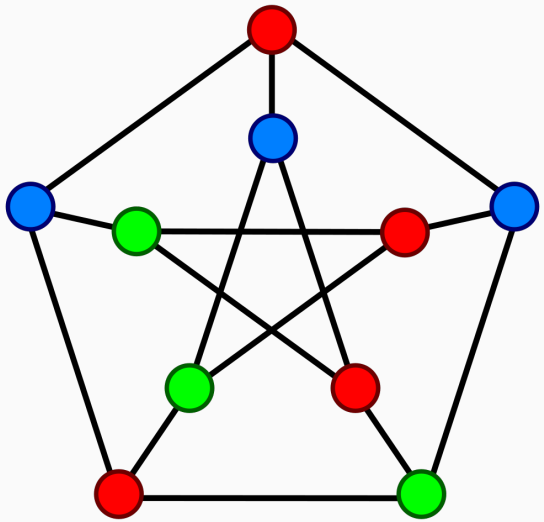
$\exists \varphi : FV(f) \to \{\text{true}, \text{false}\}$ $\quad \Phi_\varphi(f) = \text{true} \quad \iff$

per induzione

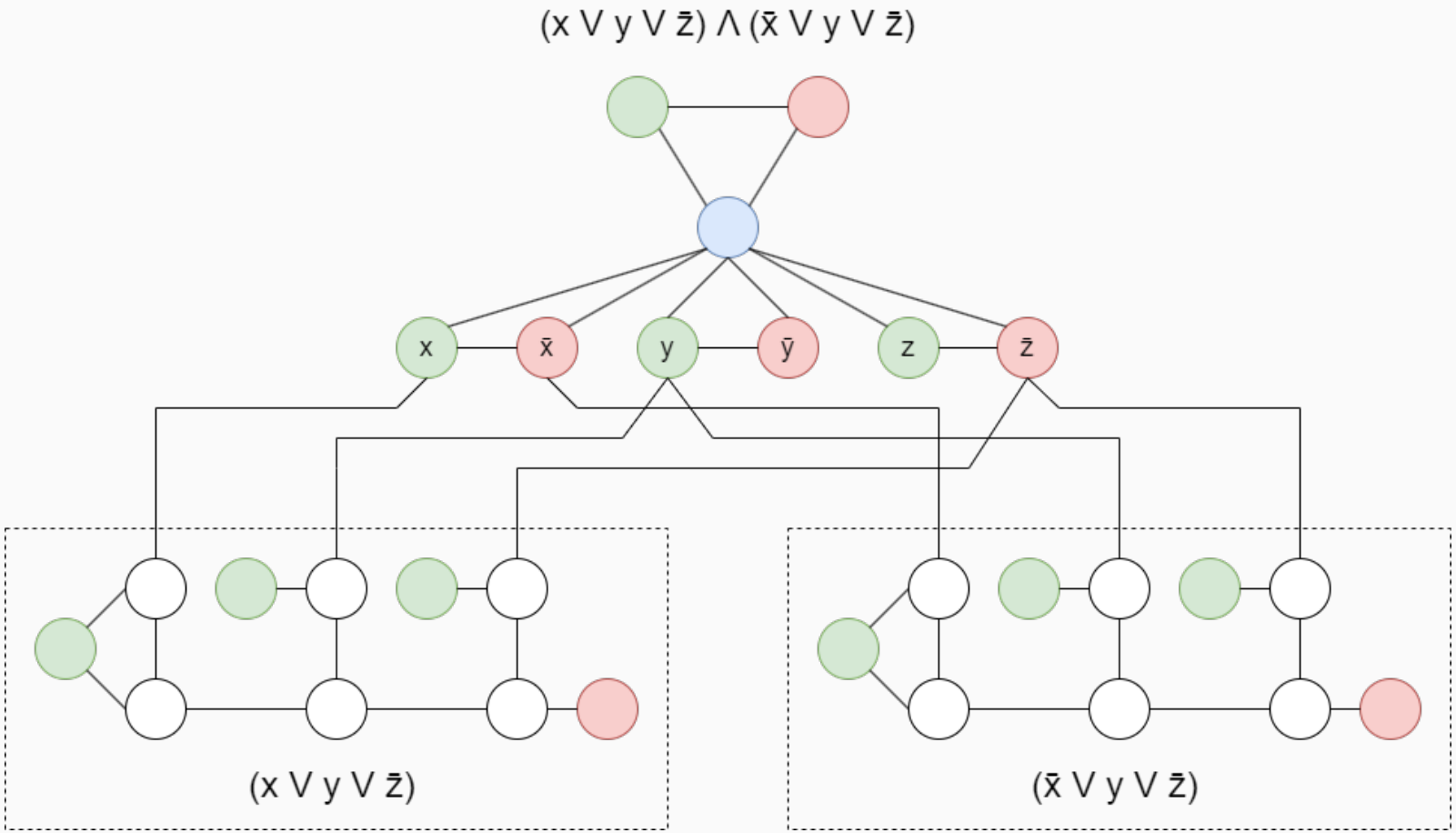
$\exists \varphi' : FV(F(f)) \to \{\text{true}, \text{false}\}$ $\quad \Phi_{\varphi'}(T(f)) = \text{true}$

3COL

Sia $G = (V, E)$ un grafo. Esiste una funzione $C : V \to \{green, red, blue\}$, tale che $C(V)$ colora i vertici V in modo che, per ogni coppia (x, y) di vertici adiacenti, $C(x) \neq C(y)$?

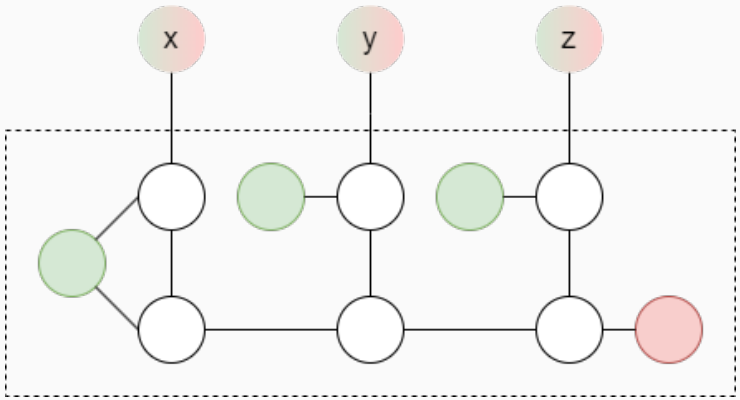


3CNF $\leq_P$ 3COL



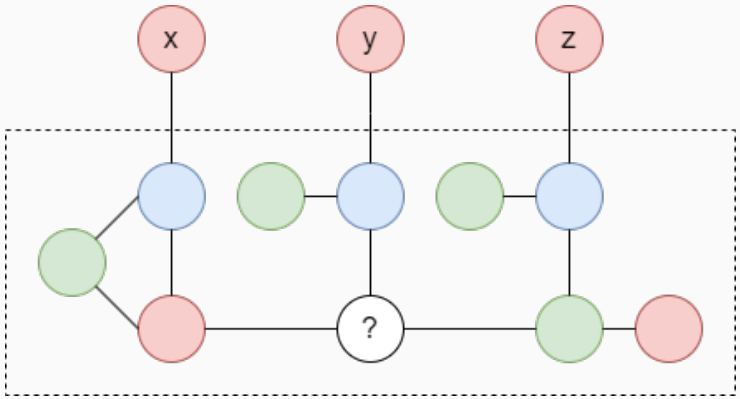
3CNF $\leq_P$ 3COL / Proprietà 1

Il sottografo è colorabile se $x = green$ o $y = green$ oppure $z = green$



3CNF $\leq_P$ 3COL / Proprietà 2

Il sottografo non è colorabile se $x = red$ e $y = red$ e $z = red$



3CNF $\leq_P$ 3COL / Dimostrazione

Sia $f \in 3cnf$

1. $\exists \varphi : FV(f) \to \{\text{true}, \text{false}\} \qquad \Phi_\varphi(f) = \text{true}$
   Per ogni clausola $x \vee y \vee z$ per ciascuna possibilità descritta dal predicato

   $$\Phi_\varphi(x) = \text{true} \quad \text{o} \quad \Phi_\varphi(y) = \text{true} \quad \text{o} \quad \Phi_\varphi(z) = \text{true}$$

   La **proprietà 1** assciura la colorabilità del sottografo
2. $\forall \varphi : FV(f) \to \{\text{true}, \text{false}\} \qquad \Phi_\varphi(f) = \text{false}$
   Esiste una clausola $x \vee y \vee z$ tale che

   $$\Phi_\varphi(x) = \Phi_\varphi(y) = \Phi_\varphi(z) = \text{false}$$

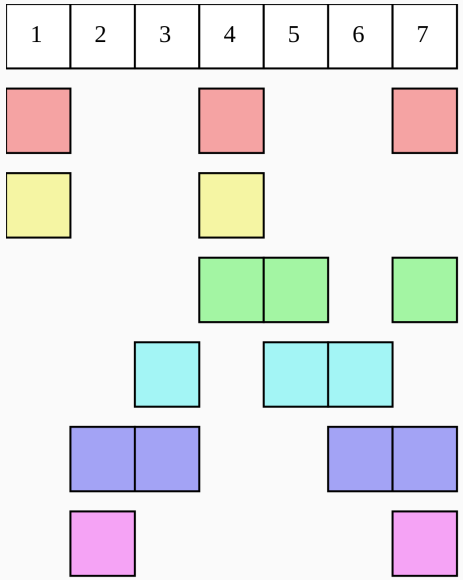
   Dalla **proprietà 2** sappiamo che il sottografo non è colorabile, quindi l'intero grafo non lo è

EXCO

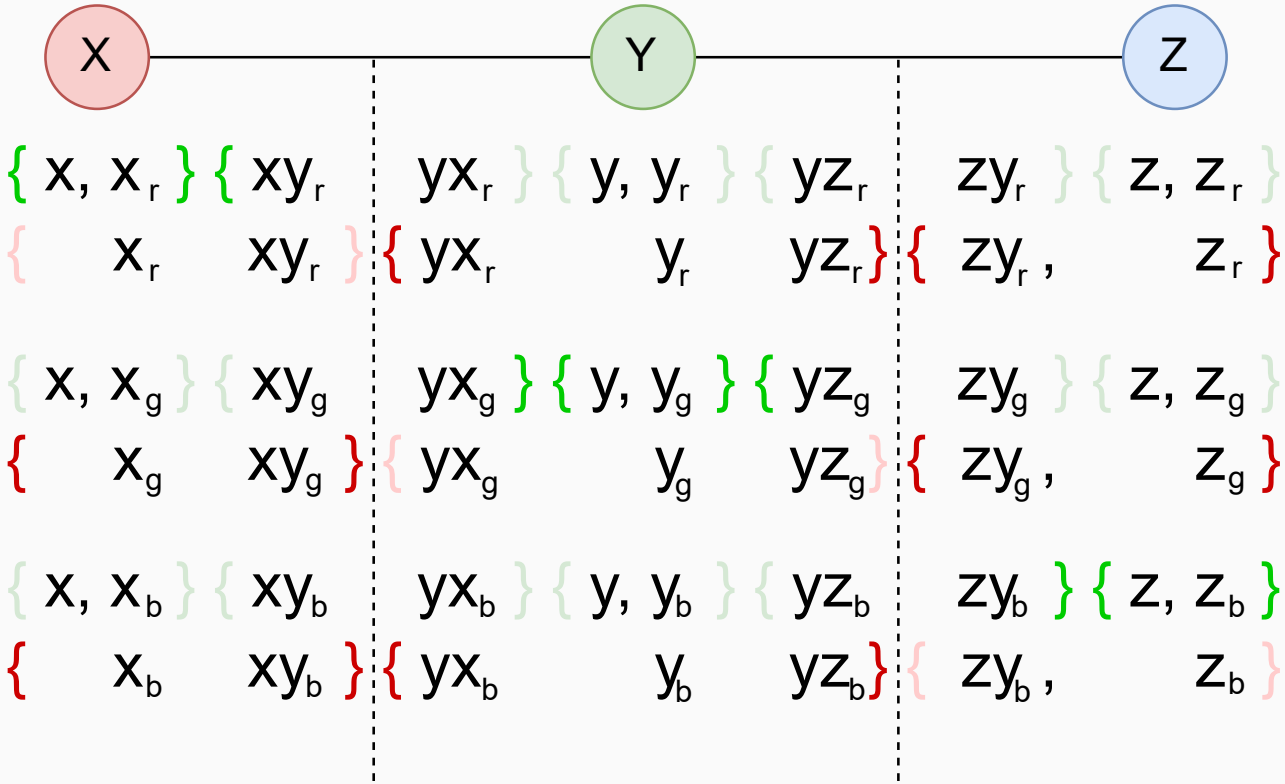
$U = \{x_1, x_2, \ldots, x_n\}$

$S = \{S_1, S_2, \ldots, S_m\} \quad con \quad S_i \subseteq U$

Esiste $C \subseteq S$ tale che $\bigcup_{S \in C} S = U$ e $\bigcap_{S \in C} S = \emptyset$?



3COL $\leq_P$ EXCO



$3\mathrm{COL} \leq_P \mathrm{EXCO}$

$$U(V,E) = V \cup \bigcup_{n \in V} \{n_r, n_g, n_b\} \cup \bigcup_{(n,m) \in E} \{nm_r, nm_g, nm_b\}$$

$$\begin{aligned}
S(V,E) = & \bigcup_{n \in V} \{\{n, n_r\}, \{n, n_g\}, \{n, n_b\}\} \cup \\
& \bigcup_{(n,m) \in E} \{nm_r\} \times \{mn_g, mn_b\} \cup \\
& \bigcup_{(n,m) \in E} \{nm_g\} \times \{mn_r, mn_b\} \cup \\
& \bigcup_{(n,m) \in E} \{nm_b\} \times \{mn_r, mn_g\} \cup \\
& \bigcup_{n \in V} \{\{n_r\} \cup \bigcup_{(n,m) \in E} \{nm_r\}\} \cup \\
& \bigcup_{n \in V} \{\{n_g\} \cup \bigcup_{(n,m) \in E} \{nm_g\}\} \cup \\
& \bigcup_{n \in V} \{\{n_b\} \cup \bigcup_{(n,m) \in E} \{nm_b\}\}
\end{aligned}$$

3COL $\leq_P$ EXCO / Dimostrazione

Sia $(V, E) \in 3col$ 3-colorabile
allora esiste $C : V \to \{r, g, b\}$ che assegna un colore ad ogni vertice in modo tale che $\forall (n, m) \in E \Rightarrow C(n) \neq C(m)$
allora esiste l'insieme

$$C = \bigcup_{n \in V} \{\{n, n_{C(n)}\}\} \cup \bigcup_{(n,m) \in E} \{nm_{C(n)}, mn_{C(m)}\} \cup \bigcup_{n \in V} \{ \bigcup_{c \in \{r,g,b\} \setminus \{C(n)\}} \{\{n_c\} \cup \bigcup_{(n,m) \in E} \{nm_c\}\}\}$$

tale che $\bigcup_{S \in C} S = U$ e $\bigcap_{S \in C} S = \emptyset$

3COL $\leq_P$ EXCO / Dimostrazione

Sia $(V, E) \in 3col$ **non** 3-colorabile

assumiamo per assurdo che esista un insieme $C \subseteq S(V, E)$ tale che $\bigcup_{S \in C} S = U$ e $\bigcap_{S \in C} S = \emptyset$

dunque $\forall n \in V \quad \exists S \in C \quad n \in S$ altrimenti $\bigcup_{S \in C} S \neq U$

quindi abbiamo $\bigcup_{n \in V} \{\{n, n_c\}\} \subseteq C$

per rispettare i vincoli di exco abbiamo dunque che

$\bigcup_{n \in V} \{\{n, n_c\}\} \cup \bigcup_{(n,m) \in c} \{\{nm_c, mn_{c'}\}\} \subseteq C$ con $c \neq c'$

dunque

$\forall (n, m) \in E \quad \{\{n, n_c\}, \{m, m_{c'}\}\}$ con $c \neq c'$

ma ciò non è possibile perchè il grafo non è 3-colorabile

Subsetsum

$S = s_1, s_2, \ldots, s_n$ $\qquad s_i \in \mathbb{N}$

$T \in \mathbb{N}$

Esiste $S' \subseteq S$ tale che $\sum_{s \in S'} = T$?

EXCO $\leq_P$ Subsetsum

$$S = \left\{ \begin{array}{c} a\ b\ \ d \\ \ \ b\ \ d\ \ f \\ a\ \ c\ \ e \\ \ \ b\ \ d \\ \ \ \ c \\ a\ \ c\ \ e\ f \end{array} \right\} \qquad S = \left\{ \begin{array}{c} 1\,1\,0\,1\,0\,0 \\ 0\,1\,0\,1\,0\,1 \\ 1\,0\,1\,0\,1\,0 \\ 0\,1\,0\,1\,0\,0 \\ 0\,0\,1\,0\,0\,0 \\ 1\,0\,1\,0\,1\,1 \end{array} \right\}$$

$$\left\{ \begin{array}{c} \ \ b\ \ d \\ a\ \ c\ \ e\ f \\ a\,b\,c\,d\,e\,f \end{array} \right\} \qquad \left\{ \begin{array}{c} 0\,1\,0\,1\,0\,0 \\ 1\,0\,1\,0\,1\,1 \\ 1\,1\,1\,1\,1\,1 \end{array} \right\}$$

EXCO $\leq_P$ Subsetsum

$$SS(U,S) = \left\{ \sum_{s_j \in U} k^j \begin{cases} 1 & s_j \in S_i \\ 0 & s_j \notin S_i \end{cases} \middle| S_i \in S \right\}$$

$$T(U,S) = \sum_{s_i \in U} k^i = \frac{k^{|U|}-1}{k-1}$$

$$k = \max_X |\{S_i \,|\, X \in S_i \in S\}|$$

EXCO $\leq_P$ Subsetsum / Dimostrazione

Sia $(U, S) \in exco$ un exact cover

allora esiste $C \subseteq S$ exact cover di (U, S)

si può dimostrare che $SS(U, C)$ è il sottoinsieme con somma T

$$\sum_{n \in SS(U,C)} n = \sum_{s_j \in U} k^j \sum_{S \in C} \begin{cases} 1 & s_j \in S \\ 0 & s_j \notin S \end{cases}$$

poichè $\bigcup_{S \in C} S = U$ e $\bigcap_{S \in C} S = \emptyset$ abbiamo che $\sum_{S \in C} \begin{cases} 1 & s_j \in S \\ 0 & s_j \notin S \end{cases} = 1$

dunque

$$\sum_{n \in SS(U,C)} n = \sum_{s_j \in U} k^j$$

EXCO $\leq_P$ Subsetsum / Dimostrazione

Sia $(U, S) \in exco$ un exact cover non valida
supponiamo per assurdo che esista $S^* \subseteq SS(U, S)$ tale che $\sum_{n \in S^*} n = \sum_{s_j \in U} k^j$ e $C \subseteq S$ exact cover di S

dunque abbiamo $\sum_{s_j \in U} k^j = \sum_{s_j \in U} k^j c_j$ con $c_j = \sum_{S \in C} \begin{cases} 1 & s_j \in S \\ 0 & s_j \notin S \end{cases}$

con $c_j \neq 1$ per qualche S

per ogni j abbiamo che $\begin{cases} c_j = 1 & c_{j+1} = 1 \\ c_j \geq k+1 & c_{j+1} = 0 \end{cases}$

ma un c_j non può essere maggiore di k perchè è il numero massimo di volte in cui $s_j \in S \in C$

Modelli quadratici

QUBO

$Q \in \mathbb{R}^{n \times n}$ matrice triangolare superiore

$x \in \{0, 1\}^n$

$$\min_x Qx^T = \min_x \sum_{i=1}^{n} \sum_{j=i}^{n} Q_{ij} x_i x_j$$

QUBO / Esempio

$$Q = \begin{pmatrix} 1 & -2 & 3 \\ 0 & 4 & -1 \\ 0 & 0 & 2 \end{pmatrix}$$

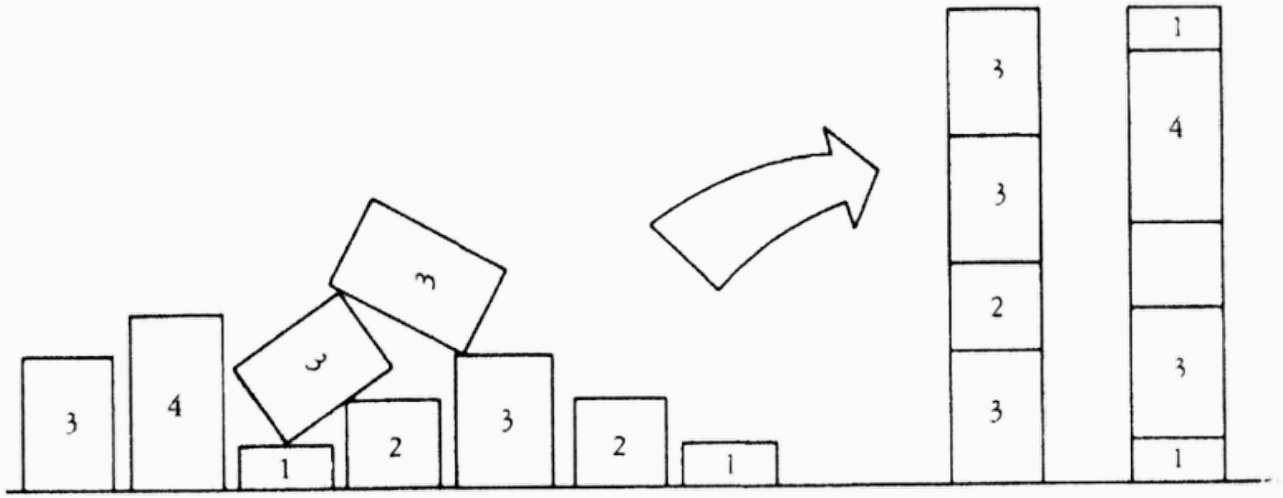
$$\min_x \left(1x_1 - 2x_1x_2 + 3x_1x_3 + 4x_2 - x_2x_3 + 2x_3\right)$$

il minimo è $x = (0, 0, 0)$

Partizionamento

$S \in \mathbb{N}^n$

Trovare $T \subseteq S$ tale che $\sum_{v \in T} v = \sum_{v \in S \setminus T} v$



Partizionamento $\leq_P$ QUBO

Possiamo osservare che $\sum_{v \in S \setminus T} v = \sum_{v \in S} v - \sum_{v \in T} v$

dunque il problema si riduce a $0 = \sum_{v \in S} v - 2 \sum_{v \in T} v$

in forma **QUBO**:

$$\min_x \left(\sum_{i=1}^{n} v_i - 2 \sum_{i=1}^{n} v_i x_i \right)^2$$

$$\min_x \left(\sum_{i=1}^{n} \sum_{j=1}^{n} v_i v_j x_i x_j - c \sum_{i=1}^{n} v_i x_i \right)$$

dove $c = \sum_{i=1}^{n} v_i$

Partizionamento $\leq_P$ QUBO

Possiamo osservare che $\sum_{v \in S \setminus T} v = \sum_{v \in S} v - \sum_{v \in T} v$

dunque il problema si riduce a $0 = \sum_{v \in S} v - 2 \sum_{v \in T} v$

in forma **QUBO**:

$$\min_x \left(c - 2 \sum_{i=1}^{n} v_i x_i \right)^2$$

$$\min_x \left(\sum_{i=1}^{n} \sum_{j=1}^{n} v_i v_j x_i x_j - c \sum_{i=1}^{n} v_i x_i \right)$$

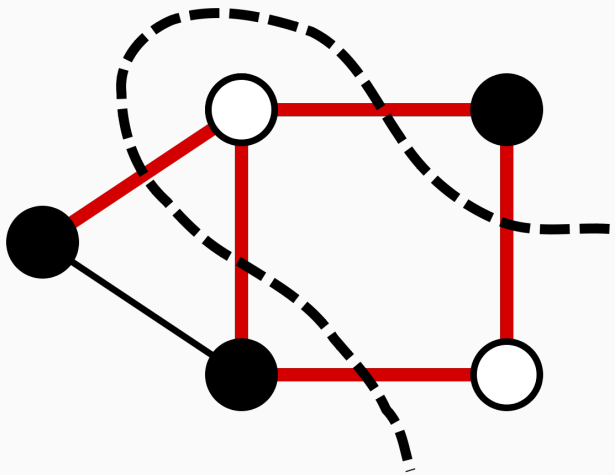
dove $c = \sum_{i=1}^{n} v_i$

Partizionamento $\leq_P$ QUBO

$$Q = \begin{pmatrix} v_1^2 - cv_1 & 2v_1v_2 & 2v_1v_3 & 2v_1v_4 & 2v_1v_5 \\ 0 & v_2^2 - cv_2 & 2v_2v_3 & 2v_2v_4 & 2v_2v_5 \\ 0 & 0 & v_3^2 - cv_3 & 2v_3v_4 & 2v_3v_5 \\ 0 & 0 & 0 & v_4^2 - cv_4 & 2v_4v_5 \\ 0 & 0 & 0 & 0 & v_5^2 - cv_5 \end{pmatrix}$$

Max cut

Sia $G = (V, E)$ un grafo
Partizionare i vertici in modo tale che il numero di archi tra le due partizioni sia massimo



Max cut $\leq_P$ QUBO

Il problema può essere formulato come

$$\max_x \sum_{i=1}^{|V|} \sum_{j=1}^{|V|} A_{ij}(x_i \text{ xor } x_j)$$

dove A è la matrice di adiacenza del grafo in forma **QUBO**:

$$\max_x \sum_{i=1}^{|V|} \sum_{j=1}^{|V|} A_{ij}(x_i + x_j - 2x_i x_j)$$

$$\max_x -2 \sum_{i=1}^{|V|} \sum_{j=1}^{|V|} A_{ij} x_i x_j + 2 \sum_{i=1}^{|V|} \sum_{j=1}^{|V|} A_{ij} x_i$$

$$\min_x \sum_{i=1}^{|V|} \sum_{j=1}^{|V|} A_{ij} x_i x_j - \sum_{i=1}^{|V|} A_i x_i \quad \text{con} \quad A_i = \sum_{j=1}^{|V|} A_{ij}$$

Max cut $\leq_P$ QUBO

$$Q = \begin{pmatrix} -A_1 & 2A_{12} & 2A_{13} & 2A_{14} & 2A_{15} \\ 0 & -A_2 & 2A_{23} & 2A_{24} & 2A_{25} \\ 0 & 0 & -A_3 & 2A_{34} & 2A_{35} \\ 0 & 0 & 0 & -A_4 & 2A_{45} \\ 0 & 0 & 0 & 0 & -A_5 \end{pmatrix}$$