Greedy e Backtracking

Marco Donadoni

Online, 17/04/2021

Algoritmo greedy

Consideriamo un problema tale per cui un algoritmo che lo risolve

- esegue un certa sequenza di passi
- ad ogni passo ha a disposizione un insieme di mosse fra cui scegliere

Algoritmo greedy

Consideriamo un problema tale per cui un algoritmo che lo risolve

- esegue un certa sequenza di passi
- ad ogni passo ha a disposizione un insieme di mosse fra cui scegliere

Algoritmo greedy

Un algoritmo si dice greedy se sceglie sempre la mossa migliore (localmente!) ad ogni passo

Algoritmo greedy

Consideriamo un problema tale per cui un algoritmo che lo risolve

- esegue un certa sequenza di passi
- ad ogni passo ha a disposizione un insieme di mosse fra cui scegliere

Algoritmo greedy

Un algoritmo si dice greedy se sceglie sempre la mossa migliore (localmente!) ad ogni passo

Ovviamente, non è sempre garantito per ogni problema che scegliere la migliore mossa permetta di ottenere la migliore soluzione globale

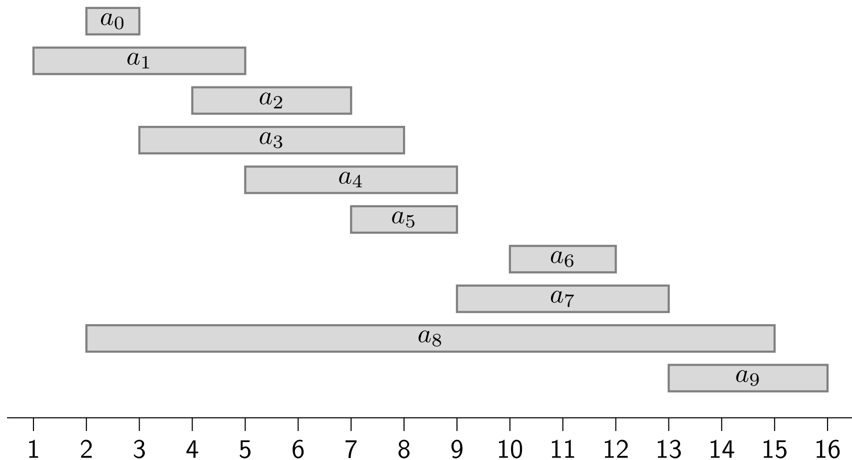
Problema di selezione delle attività

Activity-selection problem

Sia $A = \{a_0, a_1, \ldots, a_{N-1}\}$ un insieme di attività da svolgere. Ogni attività a_i ha un istante di inizio s_i e un istante di fine f_i. Due attività a_i e a_j sono compatibili se $f_i \leq s_j$ oppure $f_j \leq s_i$. Quale è il più grande sottoinsieme di A tale per cui tutte le attività sono compatibili a due a due?

Per semplicità supponiamo che $f_0 \leq f_1 \leq \cdots \leq f_{N-1}$

Problema di selezione delle attività



Problema di selezione delle attività

Definiamo A_{ij} come l'insieme di attività che iniziano dopo la fine dell'attività a_i e finiscono prima che a_j inizi

$$A_{ij} = \{a_k : s_k \geq f_i \wedge f_k \leq s_j\}$$

Problema di selezione delle attività

Definiamo A_{ij} come l'insieme di attività che iniziano dopo la fine dell'attività a_i e finiscono prima che a_j inizi

$$A_{ij} = \{a_k : s_k \geq f_i \wedge f_k \leq s_j\}$$

Sia S_{ij} la soluzione ottima del problema di selezione delle attività, considerando solo le attività in A_{ij}

Problema di selezione delle attività

Programmazione dinamica

Supponiamo che a_k faccia parte della soluzione ottima S_{ij}, ovvero $a_k \in S_{ij}$

Problema di selezione delle attività

Programmazione dinamica

Supponiamo che a_k faccia parte della soluzione ottima S_{ij}, ovvero $a_k \in S_{ij}$

L'attività a_k suddivide il problema originale in due sottoproblemi

- trovare un sottoinsieme di attività compatibili in A_{ik}
- trovare un sottoinsieme di attività compatibili in A_{kj}

Problema di selezione delle attività

Programmazione dinamica

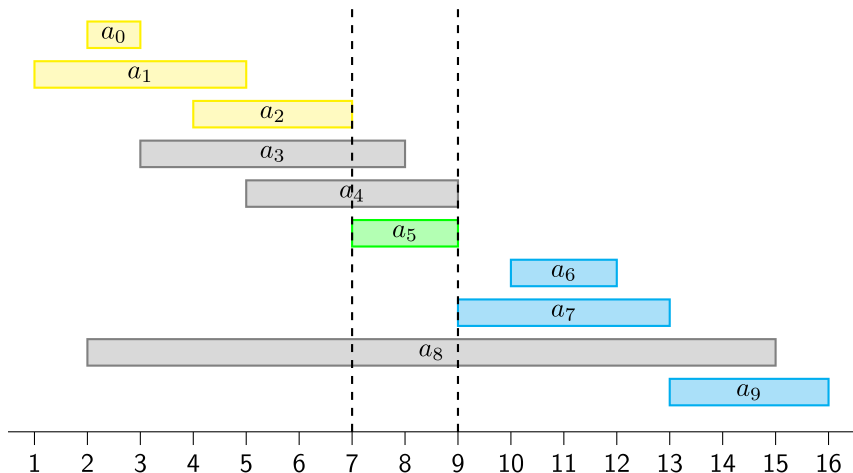
Supponiamo che a_k faccia parte della soluzione ottima S_{ij}, ovvero $a_k \in S_{ij}$

L'attività a_k suddivide il problema originale in due sottoproblemi

- trovare un sottoinsieme di attività compatibili in A_{ik}
- trovare un sottoinsieme di attività compatibili in A_{kj}

$$S_{ij} = (S_{ij} \cap A_{ik}) \cup \{a_k\} \cup (S_{ij} \cap A_{kj})$$

Problema di selezione delle attività



Problema di selezione delle attività

Programmazione dinamica

Sottostruttura ottima

Se $a_k \in S_{ij}$, allora $A_{ik} \cap S_{ij}$ è soluzione ottima del problema di selezione delle attività su A_{ik}

Problema di selezione delle attività

Programmazione dinamica

Sottostruttura ottima

Se $a_k \in S_{ij}$, allora $A_{ik} \cap S_{ij}$ è soluzione ottima del problema di selezione delle attività su A_{ik}

Dimostrazione:
Supponiamo per assurdo che $A_{ik} \cap S_{ij}$ non sia la soluzione ottima del problema per le attività A_{ik}

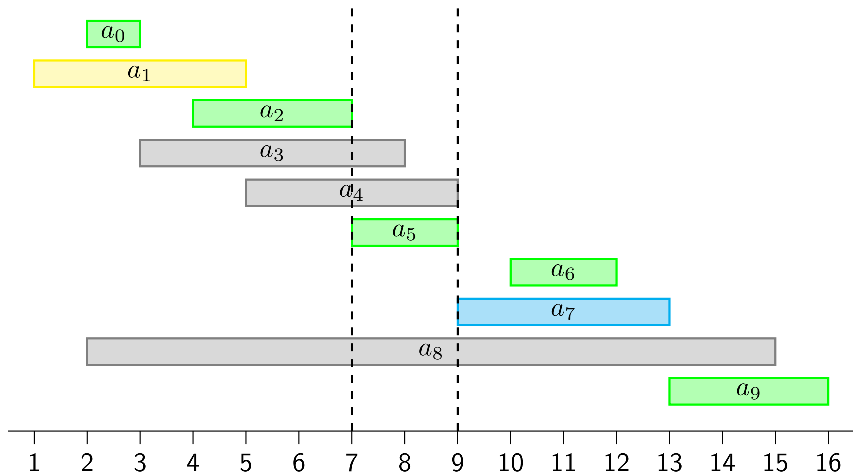
$$\implies |S_{ik}| > |A_{ik} \cap S_{ij}|$$

Possiamo sostituire S_{ik} al posto di $A_{ik} \cap S_{ij}$ ottenendo

$$\begin{aligned} |S_{ij}| &= |S_{ij} \cap A_{ik}| + |\{a_k\}| + |S_{ij} \cap A_{kj}| \\ &< |S_{ik}| \qquad\quad + |\{a_k\}| + |S_{ij} \cap A_{kj}| \end{aligned}$$

Assurdo, dato che abbiamo supposto S_{ij} essere la soluzione ottima per A_{ij}

Problema di selezione delle attività



Problema di selezione delle attività

Programmazione dinamica

Possiamo quindi risolvere il problema usando la programmazione dinamica

Sia $c[i, j] = |S_{ij}|$, allora

$$c[i, j] = \begin{cases} 0 & \text{se } A_{ij} = \emptyset \\ \max\limits_{a_k \in A_{ij}} c[i, k] + c[k, j] + 1 & \text{se } A_{ij} \neq \emptyset \end{cases}$$

Problema di selezione delle attività

Programmazione dinamica

Possiamo quindi risolvere il problema usando la programmazione dinamica

Sia $c[i,j] = |S_{ij}|$, allora

$$c[i,j] = \begin{cases} 0 & \text{se } A_{ij} = \emptyset \\ \max\limits_{a_k \in A_{ij}} c[i,k] + c[k,j] + 1 & \text{se } A_{ij} \neq \emptyset \end{cases}$$

Si può evitare di provare tutte le attività a_k in A_{ij}?

Problema di selezione delle attività

Algoritmo greedy

Consideriamo di voler costruire S_{ij} iterativamente, aggiungendo una attività alla volta in modo tale che non ci siano conflitti

Intuitivamente, potrebbe essere una buona idea scegliere sempre l'attività $a_m \in A_{ij}$ con f_m il più piccolo possibile, in modo da lasciare più tempo possibile alle altre attività dopo a_m

Problema di selezione delle attività

Algoritmo greedy

Consideriamo di voler costruire S_{ij} iterativamente, aggiungendo una attività alla volta in modo tale che non ci siano conflitti

Intuitivamente, potrebbe essere una buona idea scegliere sempre l'attività $a_m \in A_{ij}$ con f_m il più piccolo possibile, in modo da lasciare più tempo possibile alle altre attività dopo a_m

Questo è un esempio di algoritmo greedy, nel quale ad ogni passo si sceglie sempre la mossa che sembra migliore al momento. Si può dimostrare che questa strategia permette di ottenere sempre una soluzione ottima del problema.

Problema di selezione delle attività

Algoritmo greedy

Ottimalità della strategia greedy

Sia $a_m \in A_{ij}$ un'attività tale che $f_m = \min_{a_k \in A_{ij}} f_k$.
Allora esiste una soluzione ottima del problema considerando le attività A_{ij} che contiene a_m.

Problema di selezione delle attività

Algoritmo greedy

Ottimalità della strategia greedy

Sia $a_m \in A_{ij}$ un'attività tale che $f_m = \min_{a_k \in A_{ij}} f_k$.
Allora esiste una soluzione ottima del problema considerando le attività A_{ij} che contiene a_m.

Dimostrazione:
Sia S_{ij} una qualunque soluzione ottima del problema per le attività A_{ij}.

Se $a_m \in S_{ij}$ allora abbiamo finito; supponiamo quindi che $a_m \notin S_{ij}$

Sia $a_q \in S_{ij}$ l'attività con istante di fine più piccolo in S_{ij}. Per ipotesi $f_m \leq f_q$, quindi possiamo sostituire a_q con a_m in S_{ij}, ottenendo un insieme di attività compatibili con la stessa cardinalità di S_{ij}

Problema di selezione delle attività

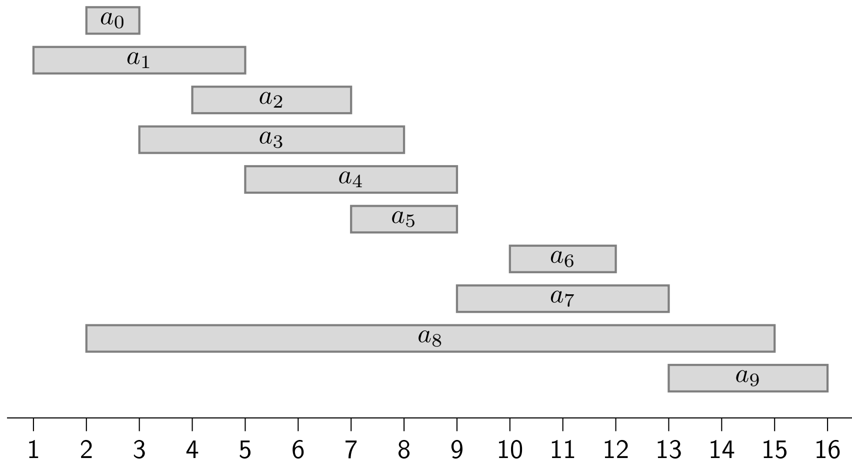
Algoritmo greedy

Sia $a_m \in A_{ij}$ un'attività tale che $f_m = \min_{a_k \in A_{ij}} f_k$

$$\begin{aligned} S_{ij} &= (S_{ij} \cap A_{im}) \ \cup \ \{a_m\} \ \cup \ (S_{ij} \cap A_{mj}) \\ &= \{a_m\} \ \cup \ (S_{ij} \cap A_{mj}) \\ &= \{a_m\} \ \cup \ S_{mj} \end{aligned}$$

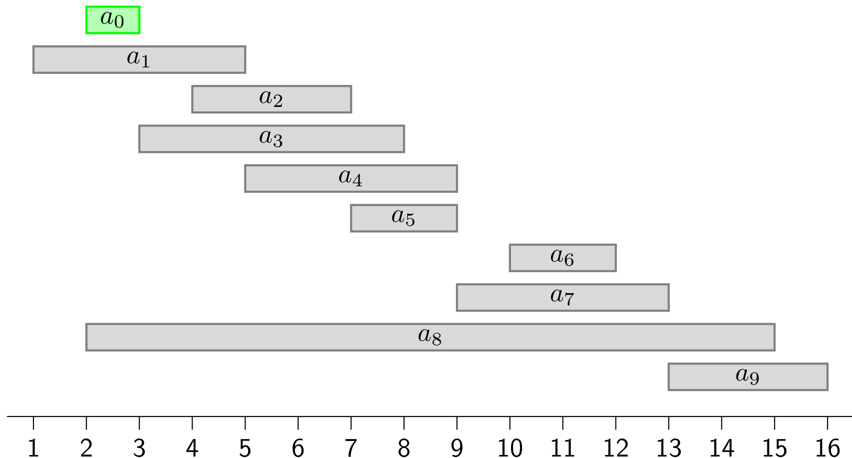
Problema di selezione delle attività

Algoritmo greedy



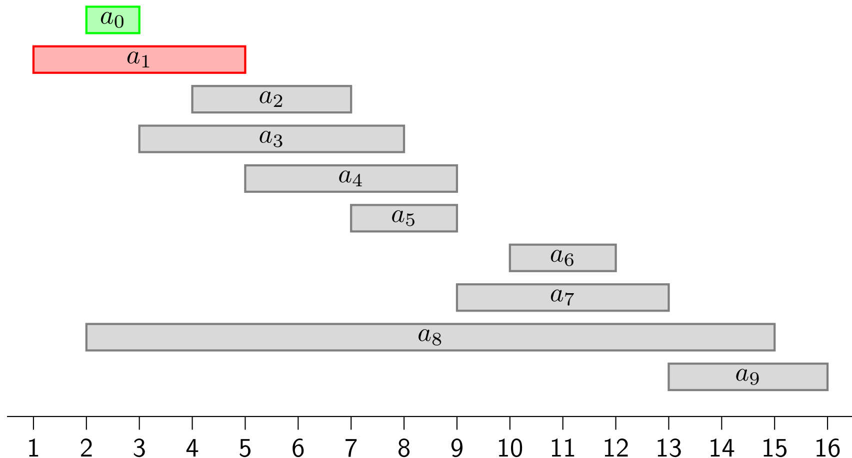
Problema di selezione delle attività

Algoritmo greedy



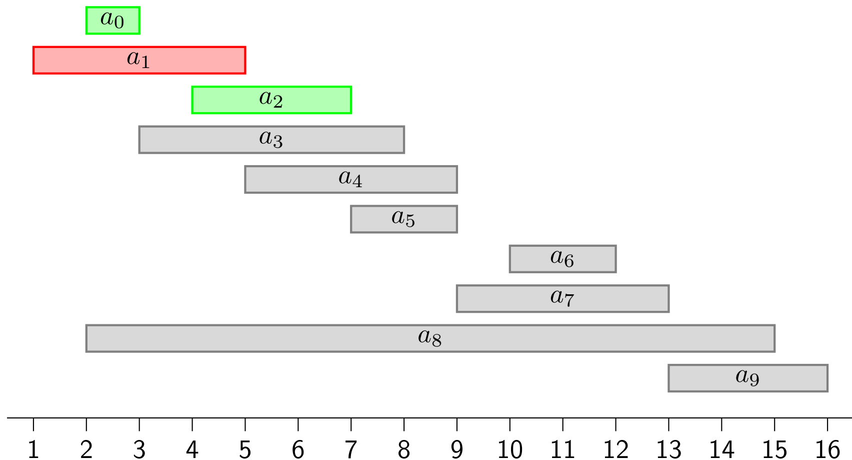
Problema di selezione delle attività

Algoritmo greedy



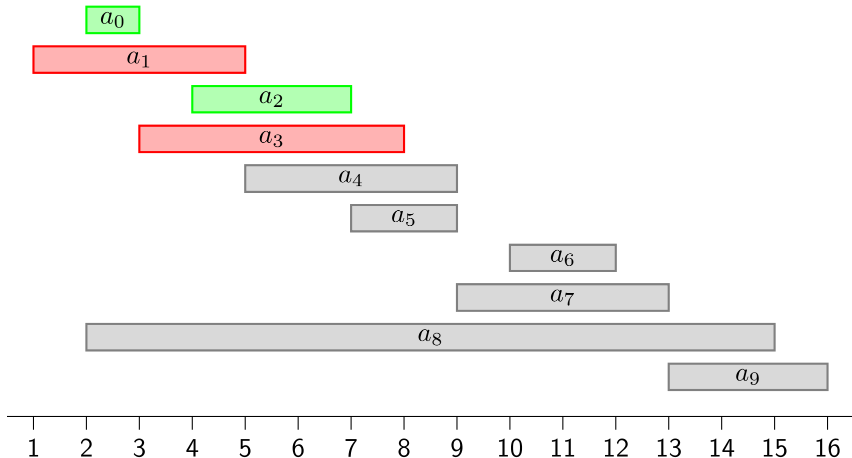
Problema di selezione delle attività

Algoritmo greedy



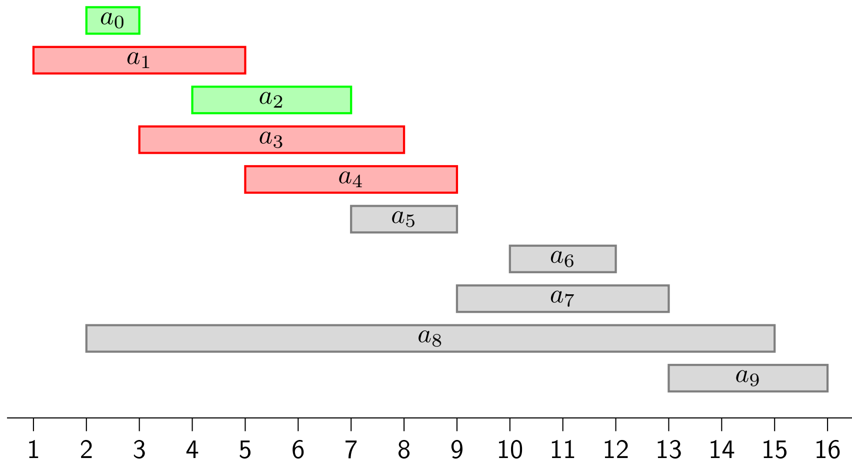
Problema di selezione delle attività

Algoritmo greedy



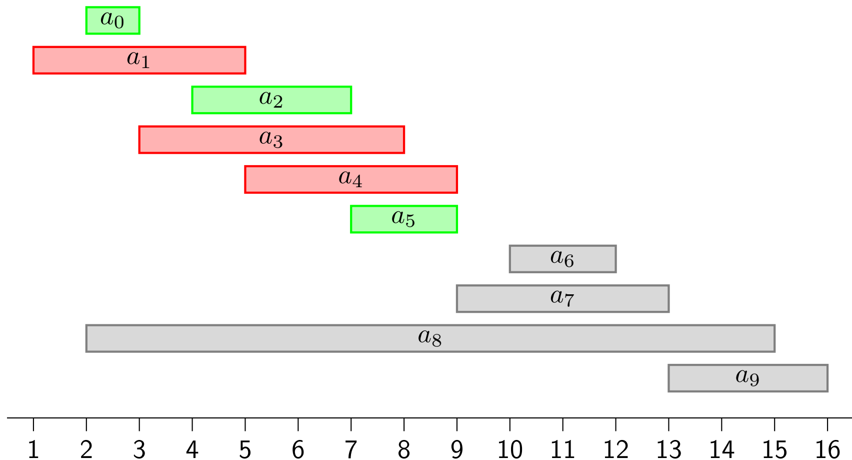
Problema di selezione delle attività

Algoritmo greedy



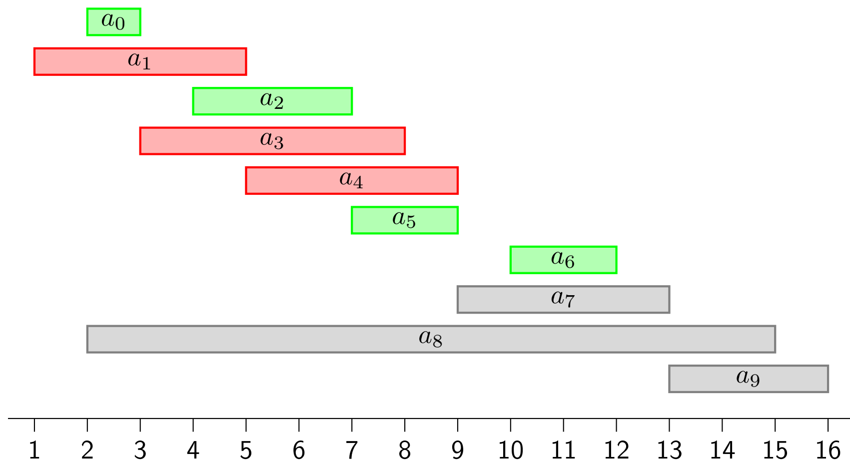
Problema di selezione delle attività

Algoritmo greedy



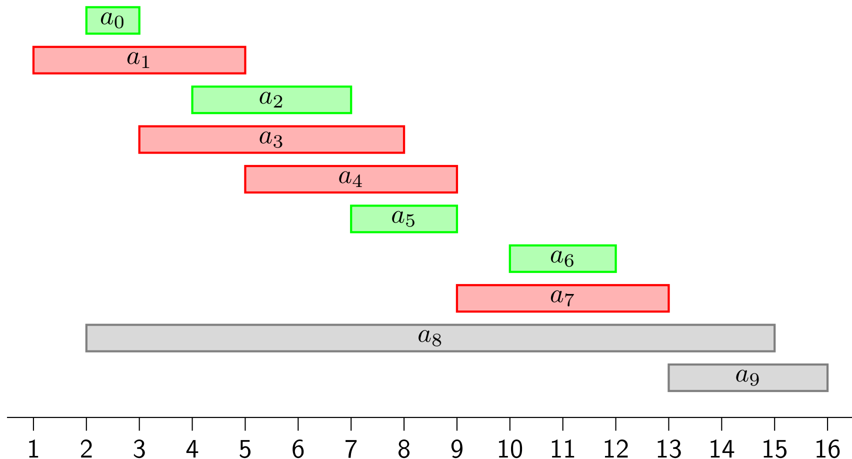
Problema di selezione delle attività

Algoritmo greedy



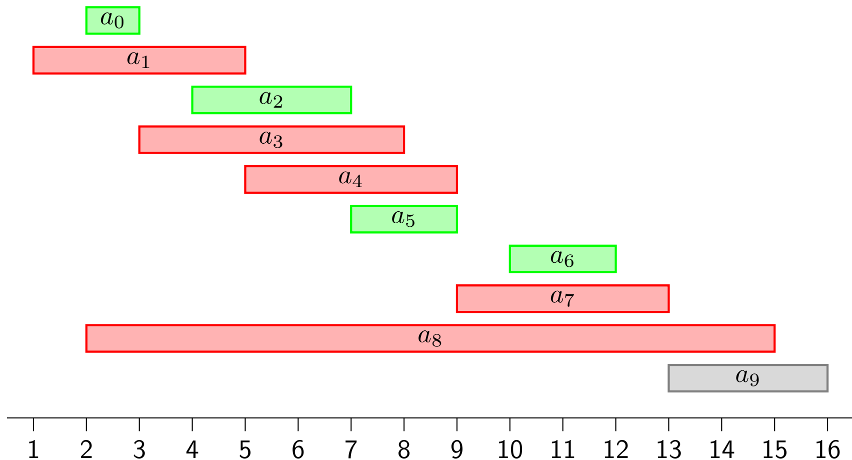
Problema di selezione delle attività

Algoritmo greedy



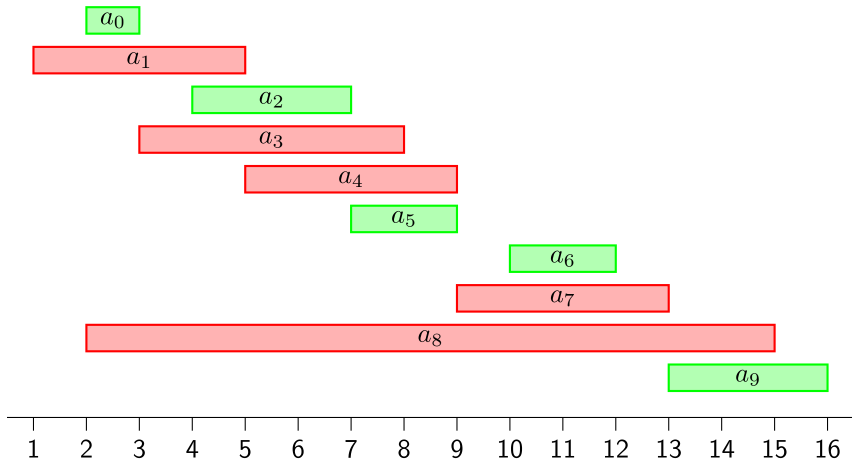
Problema di selezione delle attività

Algoritmo greedy



Problema di selezione delle attività

Algoritmo greedy



Problema di selezione delle attività

Algoritmo greedy

```
1  // Abbiamo supposto che f[0] <= f[1] <= ... <= f[N-1]
2  int activity_selection(int N, int s[], int f[]) {
3     int k = 0;
4     int sol = 1;
5     for (int i = 1; i < N; i++) {
6        if (s[i] >= f[k]) {
7           sol++;
8           k = i;
9        }
10    }
11    return sol;
12 }
```

Backtracking

Soluzione ammissibile

Dato un problema, una soluzione è *ammissibile* se soddisfa un certo insieme di proprietà

Ad esempio, nel problema dello zaino è necessario che l'insieme degli oggetti scelti abbia peso inferiore alla capacità dello zaino

Backtracking

Soluzione ammissibile

Dato un problema, una soluzione è *ammissibile* se soddisfa un certo insieme di proprietà

Ad esempio, nel problema dello zaino è necessario che l'insieme degli oggetti scelti abbia peso inferiore alla capacità dello zaino

In alcuni problemi bisogna considerare tutte le possibili soluzioni ammissibili, ad esempio in problemi in cui viene chiesta l'enumerazione o il conteggio di tutte le soluzioni possibili

Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

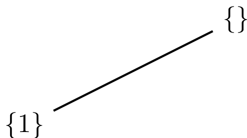
Esempio: permutazioni dei numeri da 1 a 3

$$\{\}$$

Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

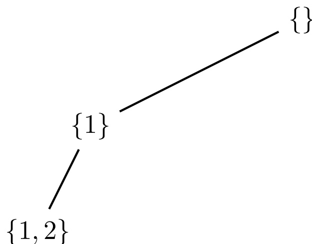
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

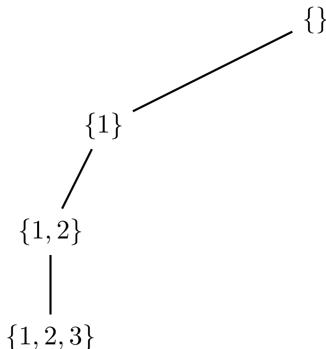
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

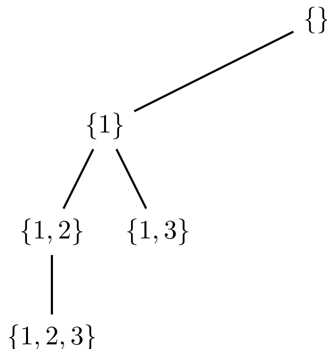
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

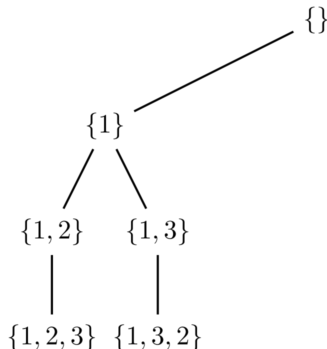
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

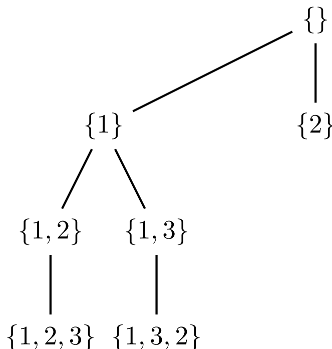
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

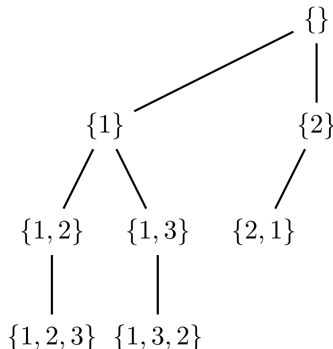
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

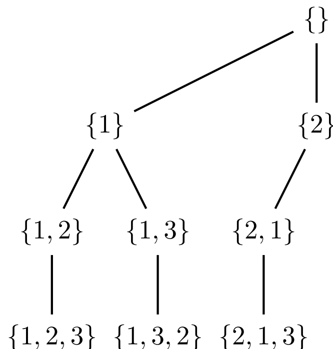
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

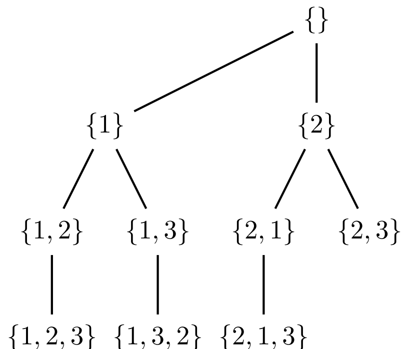
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

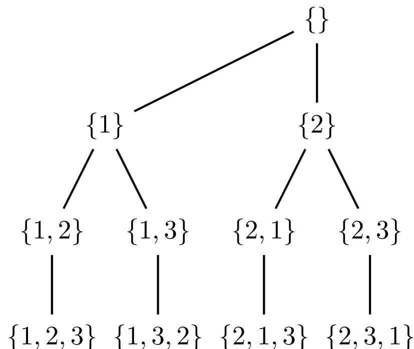
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

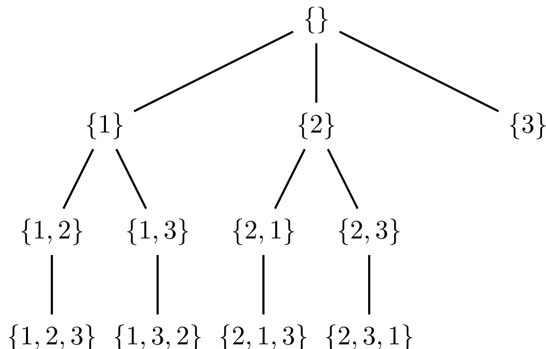
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

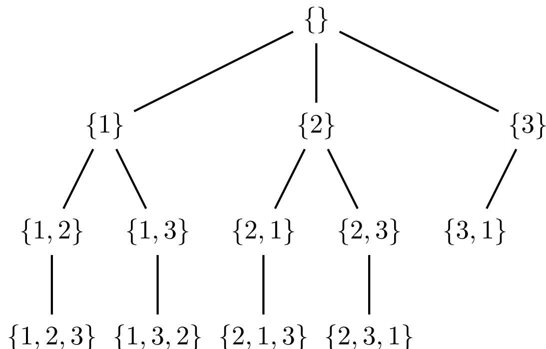
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

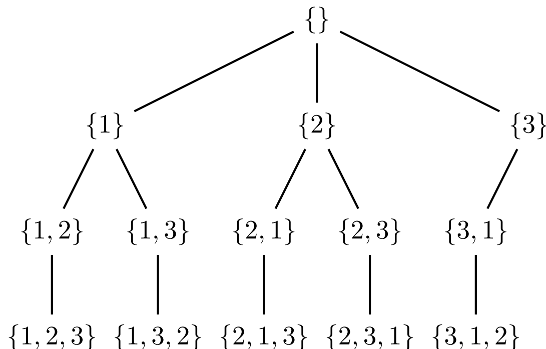
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

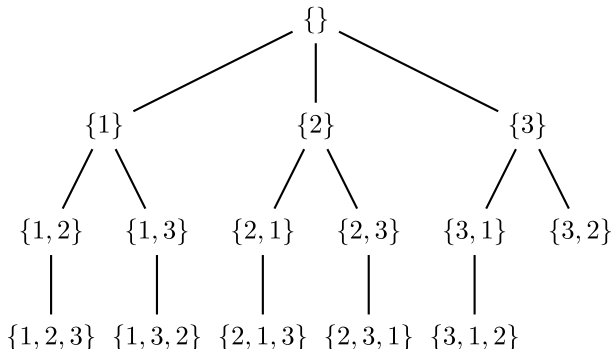
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

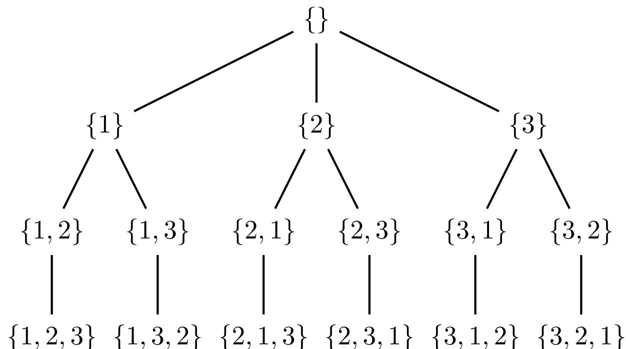
Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Idea: formuliamo il problema come una visita su un albero, dove ogni nodo interno rappresenta una soluzione parziale e le foglie rappresentano le soluzioni ammissibili

Esempio: permutazioni dei numeri da 1 a 3



Backtracking

Il backtracking è un "prototipo" di algoritmo che permette di trovare tutte le soluzioni ammissibili di un problema.

Per utilizzare la tecnica del backtracking, un problema viene descritto in termini di

- `state_t initial_state`
  uno stato iniziale

Backtracking

Il backtracking è un “prototipo” di algoritmo che permette di trovare tutte le soluzioni ammissibili di un problema.

Per utilizzare la tecnica del backtracking, un problema viene descritto in termini di

- `state_t initial_state`
  uno stato iniziale
- `bool is_valid(state_t)`
  funzione che controlla se uno stato è valido

Backtracking

Il backtracking è un "prototipo" di algoritmo che permette di trovare tutte le soluzioni ammissibili di un problema.

Per utilizzare la tecnica del backtracking, un problema viene descritto in termini di

- `state_t initial_state`
  uno stato iniziale
- `bool is_valid(state_t)`
  funzione che controlla se uno stato è valido
- `bool is_solution(state_t)`
  funzione che controlla se uno stato è soluzione del problema

Backtracking

Il backtracking è un “prototipo” di algoritmo che permette di trovare tutte le soluzioni ammissibili di un problema.

Per utilizzare la tecnica del backtracking, un problema viene descritto in termini di

- `state_t initial_state`
  uno stato iniziale
- `bool is_valid(state_t)`
  funzione che controlla se uno stato è valido
- `bool is_solution(state_t)`
  funzione che controlla se uno stato è soluzione del problema
- `void process_solution(state_t)`
  funzione che processa una soluzione

Backtracking

Il backtracking è un "prototipo" di algoritmo che permette di trovare tutte le soluzioni ammissibili di un problema.

Per utilizzare la tecnica del backtracking, un problema viene descritto in termini di

- state_t initial_state
  uno stato iniziale
- bool is_valid(state_t)
  funzione che controlla se uno stato è valido
- bool is_solution(state_t)
  funzione che controlla se uno stato è soluzione del problema
- void process_solution(state_t)
  funzione che processa una soluzione
- vector<action_t> actions(state_t)
  funzione che calcola le azioni possibili in uno stato

Backtracking

Il backtracking è un "prototipo" di algoritmo che permette di trovare tutte le soluzioni ammissibili di un problema.

Per utilizzare la tecnica del backtracking, un problema viene descritto in termini di

- `state_t initial_state`
  uno stato iniziale
- `bool is_valid(state_t)`
  funzione che controlla se uno stato è valido
- `bool is_solution(state_t)`
  funzione che controlla se uno stato è soluzione del problema
- `void process_solution(state_t)`
  funzione che processa una soluzione
- `vector<action_t> actions(state_t)`
  funzione che calcola le azioni possibili in uno stato
- `state_t next_state(state_t, action_t)`
  funzione che calcola il prossimo stato, dato lo stato corrente e una azione

Backtracking

```
1  void backtracking(state_t state) {
2     if (!is_valid(state)) {
3        // stato non valido
4        return;
5     }
6
7     if (is_solution(state)) {
8        // processa la soluzione
9        process_solution(state);
10       return;
11    }
12
13    for (action_t action : actions(state)) {
14       // per ogni possibile azione, calcola il prossimo stato
15       // e chiama ricorsivamente `backtracking`
16       backtracking(next_state(state, action));
17    }
18 }
19
20 // chiamo backtracking con lo stato iniziale
21 backtracking(initial_state);
```

Backtracking

Consideriamo il problema di elencare tutte le permutazioni dei numeri da 1 a N

```
1  typedef vector<int> state_t; // lo stato e' un vettore di interi
2  typedef int action_t; // un'azione e' un intero
3
4  int N;
5  state_t initial_state; // lo stato iniziale e' un vettore vuoto
6
7  bool is_valid(state_t state) {
8     // controlla che state contenga solo numeri da 1 a N,
9     // ciascun numero una sola volta
10    vector<int> count(N);
11    for (int x : state) {
12       if (x < 1 || x > N) return false;
13       count[x-1]++;
14       if (count[x-1] > 1) return false;
15    }
16    return true;
17 }
18
19 bool is_solution(state_t state) {
20    return state.size() == N;
21 }
```

Backtracking

```
void process_solution(state_t state) {
   for (int x : state) {
      cout << x << " ";
   }
   cout << endl;
}

vector<action_t> actions(state_t state) {
   // ritorna tutti i numeri da 1 a N
   // nota: si puo' fare di meglio!
   vector<action_t> a(N);
   iota(a.begin(), a.end(), 1);
   return a;
}

state_t next_state(state_t state, action_t action) {
   // aggiungi il numero alla soluzione parziale
   state.push_back(action);
   return state;
}
```

Nota: questo algoritmo si può implementare molto più efficientemente!

Backtracking

L'algoritmo di backtracking presentato è solo un prototipo generale e possono esserci moltissime variazioni, ad esempio

- mantenere un unico stato globale, che viene modificato prima e dopo ogni chiamata ricorsiva a `backtracking`
- implementare `actions` in modo che nessuna azione possa portare ad uno stato non valido; in questo caso non serve più la funzione `is_valid`
- implementare l'algoritmo in maniera più compatta, senza utilizzare le funzioni mostrate
- se ci sono stati ripetuti, memorizzare dei risultati intermedi (programmazione dinamica)
- ottimizzazioni ad-hoc per il problema da risolvere
- ...

Backtracking

Diversa implementazione del problema precedente

```
1  int N;
2  vector<int> perm; // un unico stato globale
3  vector<bool> taken;
4
5  void backtracking() {
6     if (perm.size() == N) {
7        // ho trovato una permutazione valida
8        for (int x : perm)
9           cout << x << " ";
10       cout << endl;
11       return;
12    }
13
14    for (int i = 1; i <= N; i++) {
15       if (!taken[i-1]) {
16          // aggiungo i alla permutazione
17          taken[i-1] = true;
18          perm.push_back(i);
19          // chiamata ricorsiva
20          backtracking();
21          // tolgo i dalla permutazione
22          taken[i-1] = false;
23          perm.pop_back();
24       }
25    }
26 }
```

Backtracking (ottimizzazione)

In alcuni problemi, non tutte le soluzioni ammissibili sono equivalenti ma alcune soluzioni sono migliori di altre. Ad esempio, nel problema dello zaino, una soluzione ammissibile è migliore di un'altra se il valore complessivo degli oggetti inseriti nello zaino è maggiore.

Come facciamo a considerare queste differenze?

Backtracking (ottimizzazione)

In alcuni problemi, non tutte le soluzioni ammissibili sono equivalenti ma alcune soluzioni sono migliori di altre. Ad esempio, nel problema dello zaino, una soluzione ammissibile è migliore di un'altra se il valore complessivo degli oggetti inseriti nello zaino è maggiore.

Come facciamo a considerare queste differenze?

- Assegniamo un costo ad ogni azione e cerchiamo il percorso di lunghezza minima dallo stato iniziale ad una qualunque soluzione ammissibile

Backtracking (ottimizzazione)

In alcuni problemi, non tutte le soluzioni ammissibili sono equivalenti ma alcune soluzioni sono migliori di altre. Ad esempio, nel problema dello zaino, una soluzione ammissibile è migliore di un'altra se il valore complessivo degli oggetti inseriti nello zaino è maggiore.

Come facciamo a considerare queste differenze?

- Assegniamo un costo ad ogni azione e cerchiamo il percorso di lunghezza minima dallo stato iniziale ad una qualunque soluzione ammissibile
- Assegniamo un valore ad ogni soluzione ammissibile e cerchiamo la soluzione con valore massimo (o minimo)

Backtracking (ottimizzazione)

Modifichiamo l'algoritmo di backtracking per cercare una delle soluzioni migliori fra quelle ammissibili.

Abbiamo bisogno di:

- `int f(state_t)`
  funzione che calcola il valore di una soluzione
- salvare la soluzione migliore trovata

Backtracking (ottimizzazione)

```
1  int best_value = -INF;
2  state_t best_solution;
3
4  void backtracking(state_t state) {
5      if (!is_valid(state)) {
6          // stato non valido
7          return;
8      }
9
10     if (is_solution(state)) {
11         int value = f(state);
12         if (value > best_value) {
13             // ho trovato una soluzione migliore
14             best_value = value;
15             best_solution = state;
16         }
17         return;
18     }
19
20     for (action_t action : actions(state)) {
21         // per ogni possibile azione, calcola il prossimo stato
22         // e chiama ricorsivamente ‘backtracking‘
23         backtracking(next_state(state, action));
24     }
25 }
```

Branch and Bound

La tecnica di branch and bound permette di velocizzare l’algoritmo di backtracking, evitando di esplorare parte dello spazio di ricerca.

Branch and Bound

Consideriamo una funzione $u(x)$ tale per cui ogni soluzione y raggiungibile dallo stato x abbia $f(y) \leq u(x)$, ovvero $u(x)$ è upper bound del valore di ogni possibile soluzione raggiungibile da x.

Branch and Bound

Consideriamo una funzione $u(x)$ tale per cui ogni soluzione y raggiungibile dallo stato x abbia $f(y) \leq u(x)$, ovvero $u(x)$ è upper bound del valore di ogni possibile soluzione raggiungibile da x.

Supponiamo, durante l'esecuzione dell'algoritmo di backtracking, di essere in un stato B; indichiamo invece con A la soluzione migliore trovata fino ad ora.

Cosa succede se $u(B) \leq f(A)$?

Branch and Bound

Consideriamo una funzione $u(x)$ tale per cui ogni soluzione y raggiungibile dallo stato x abbia $f(y) \leq u(x)$, ovvero $u(x)$ è upper bound del valore di ogni possibile soluzione raggiungibile da x.

Supponiamo, durante l'esecuzione dell'algoritmo di backtracking, di essere in un stato B; indichiamo invece con A la soluzione migliore trovata fino ad ora.

Cosa succede se $u(B) \leq f(A)$?
In questo caso, non serve visitare lo stato B dato che ogni soluzione z raggiungibile da B avrà

$$f(z) \leq u(B) \leq f(A)$$

Branch and Bound

La funzione $u(x)$ deve avere le seguenti proprietà

- per ogni soluzione y, se $x \rightsquigarrow y$ allora $f(y) \leq u(x)$
- deve essere "veloce" da calcolare, più veloce di visitare tutti gli stati raggiungibili da x
- non deve sovrastimare di molto $\max_{y : x \rightsquigarrow y} f(y)$

Branch and Bound

La funzione $u(x)$ deve avere le seguenti proprietà

- per ogni soluzione y, se $x \rightsquigarrow y$ allora $f(y) \leq u(x)$
- deve essere "veloce" da calcolare, più veloce di visitare tutti gli stati raggiungibili da x
- non deve sovrastimare di molto $\max_{y : x \rightsquigarrow y} f(y)$

Come trovo $u(x)$?

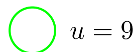
Branch and Bound

La funzione $u(x)$ deve avere le seguenti proprietà

- per ogni soluzione y, se $x \rightsquigarrow y$ allora $f(y) \leq u(x)$
- deve essere "veloce" da calcolare, più veloce di visitare tutti gli stati raggiungibili da x
- non deve sovrastimare di molto $\max_{y : x \rightsquigarrow y} f(y)$

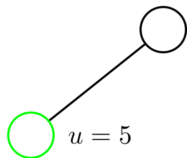
Come trovo $u(x)$? Di solito, rilassando le condizioni imposte dal problema. Ad esempio, nel problema dello zaino, si può considerare la variante del problema dello zaino frazionario, che può essere risolta velocemente con un algoritmo greedy.

Branch and Bound



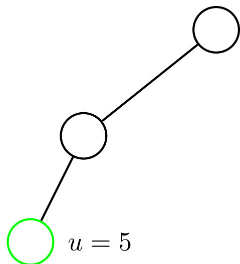
Migliore soluzione: $-\infty$

Branch and Bound



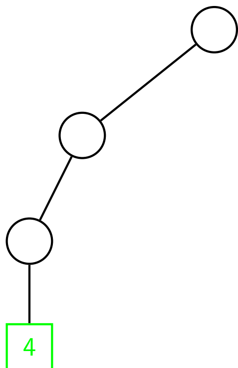
Migliore soluzione: $-\infty$

Branch and Bound



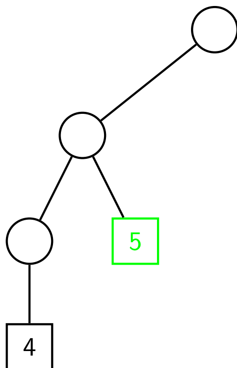
Migliore soluzione: $-\infty$

Branch and Bound



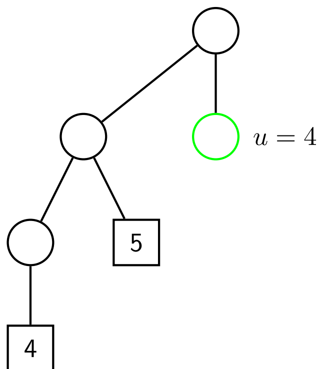
Migliore soluzione: 4

Branch and Bound



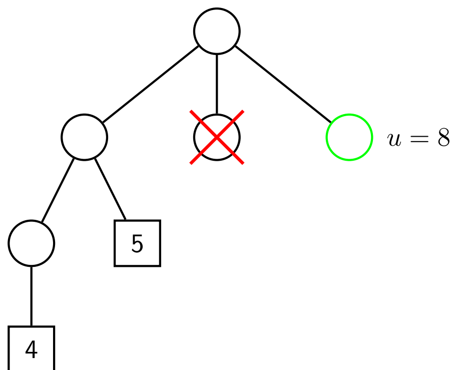
Migliore soluzione: 5

Branch and Bound



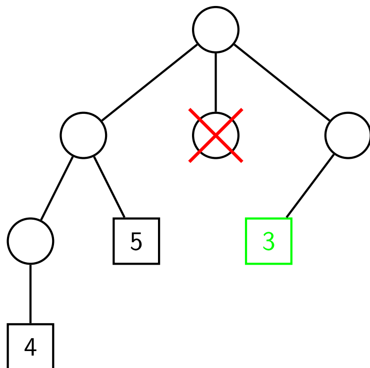
Migliore soluzione: 5

Branch and Bound



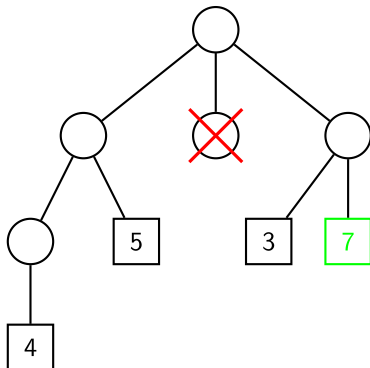
Migliore soluzione: 5

Branch and Bound



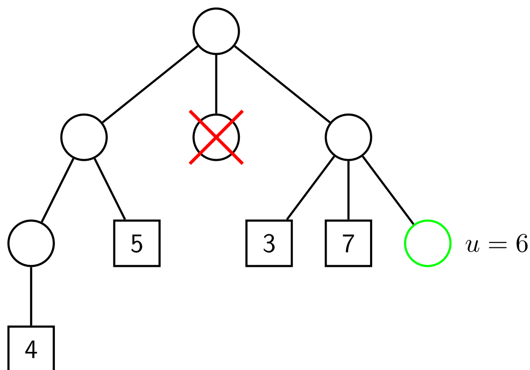
Migliore soluzione: 5

Branch and Bound



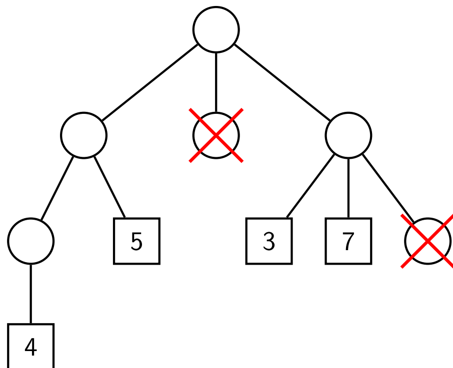
Migliore soluzione: 7

Branch and Bound



Migliore soluzione: 7

Branch and Bound



Migliore soluzione: 7

Branch and Bound

```
1  int best_value = -INF;
2  state_t best_solution;
3
4  void backtracking(state_t state) {
5      // stato non valido
6      if (!is_valid(state)) return;
7      // controllo l'upper bound
8      if (u(state) < best_value) return;
9
10     if (is_solution(state)) {
11         int value = f(state);
12         if (value > best_value) {
13             // ho trovato una soluzione migliore
14             best_value = value;
15             best_solution = state;
16         }
17         return;
18     }
19
20     for (action_t action : actions(state)) {
21         // per ogni possibile azione, calcola il prossimo stato
22         // e chiama ricorsivamente `backtracking`
23         backtracking(next_state(state, action));
24     }
25 }
```

Algoritmo Minimax

Consideriamo il problema di trovare la strategia migliore in un gioco competitivo con due giocatori.

Algoritmo Minimax

Consideriamo il problema di trovare la strategia migliore in un gioco competitivo con due giocatori.

Come in precedenza, assegniamo ad ogni stato finale della partita un valore.
I due giocatori hanno obiettivi opposti

- Il giocatore MAX punta a massimizzare il valore dello stato finale della partita
- Il giocatore MIN punta a minimizzare il valore dello stato finale della partita

Algoritmo Minimax

Consideriamo il problema di trovare la strategia migliore in un gioco competitivo con due giocatori.

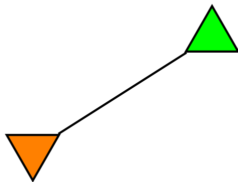
Come in precedenza, assegniamo ad ogni stato finale della partita un valore.
I due giocatori hanno obiettivi opposti

- Il giocatore MAX punta a massimizzare il valore dello stato finale della partita
- Il giocatore MIN punta a minimizzare il valore dello stato finale della partita

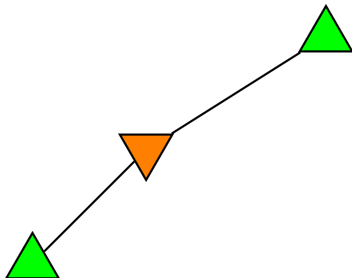
Con un approccio molto simile al backtacking, l'algoritmo minimax permette di calcolare la strategia migliore da seguire, simulando tutte le partite possibili.

Algoritmo MiniMax

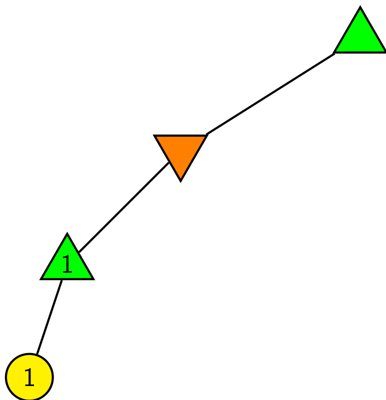
Algoritmo MiniMax



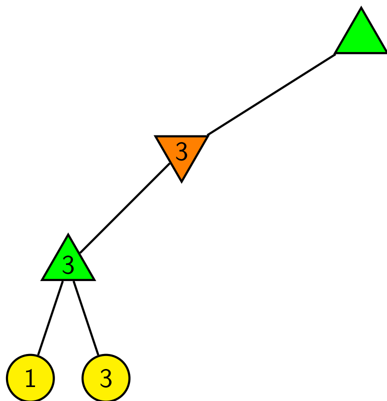
Algoritmo MiniMax



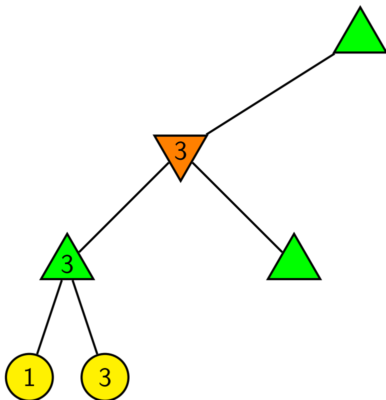
Algoritmo MiniMax



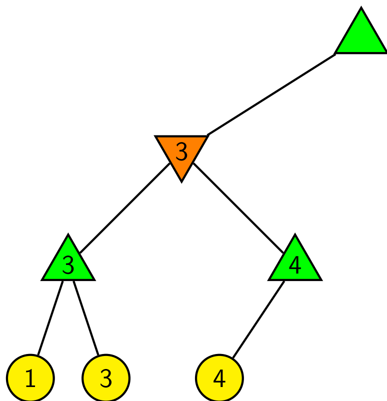
Algoritmo MiniMax



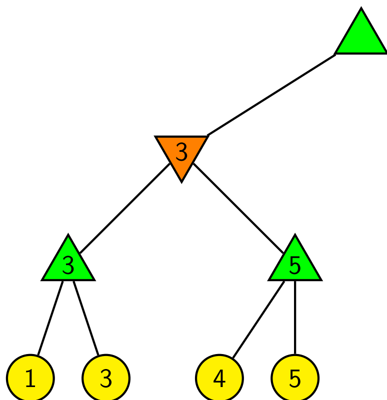
Algoritmo MiniMax



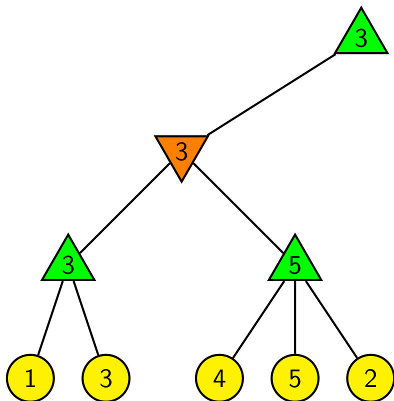
Algoritmo MiniMax



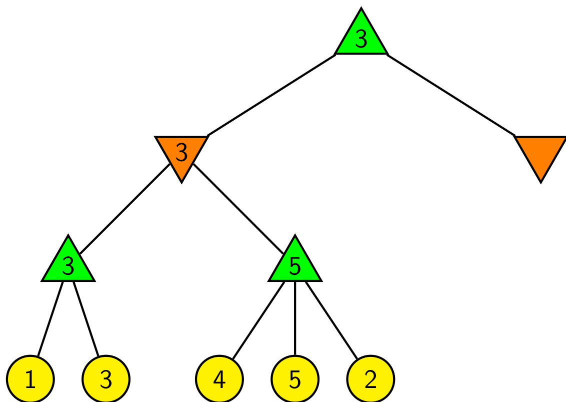
Algoritmo MiniMax



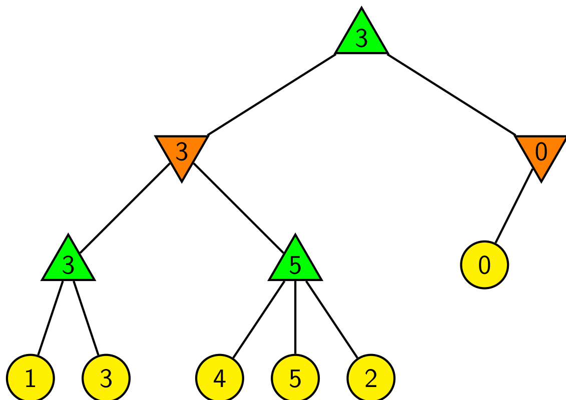
Algoritmo MiniMax



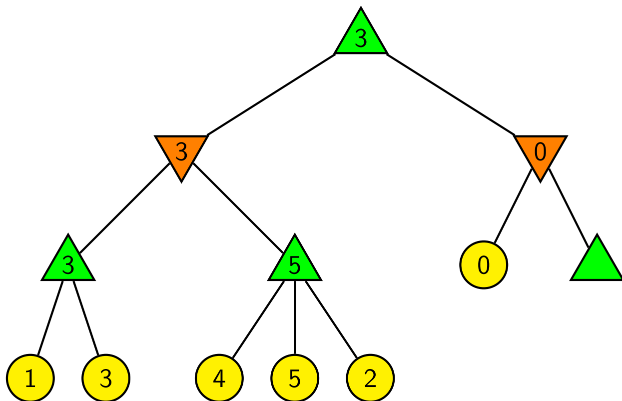
Algoritmo MiniMax



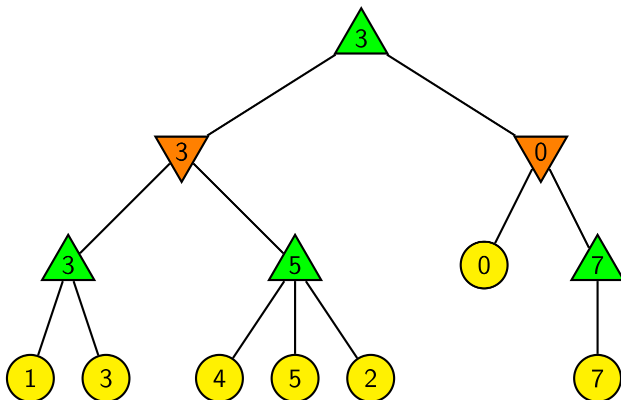
Algoritmo MiniMax



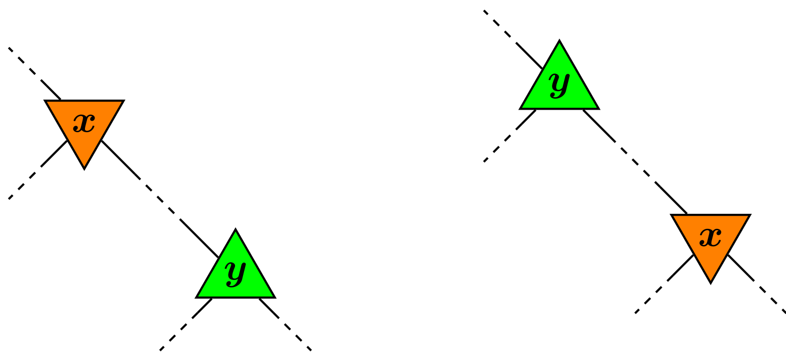
Algoritmo MiniMax



Algoritmo MiniMax



Alpha-Beta pruning



Se $x \leq y$, è inutile continuare ad espandere il nodo in basso

Alpha-Beta pruning

L'algoritmo associa ad ogni nodo due valori α e β, che vengono propagati da un nodo verso i suoi nodi figli

- α è il punteggio minimo che il giocatore MAX può ottenere in uno dei nodi antenati
- β è il punteggio massimo che il giocatore MIN può ottenere in uno dei nodi antenati

Alpha-Beta pruning

L'algoritmo associa ad ogni nodo due valori α e β, che vengono propagati da un nodo verso i suoi nodi figli

- α è il punteggio minimo che il giocatore MAX può ottenere in uno dei nodi antenati
- β è il punteggio massimo che il giocatore MIN può ottenere in uno dei nodi antenati

I valori α e β vengono aggiornati durante la ricerca, rispettivamente dai nodi "max" e dai nodi "min".

Alpha-Beta pruning

L'algoritmo associa ad ogni nodo due valori α e β, che vengono propagati da un nodo verso i suoi nodi figli

- α è il punteggio minimo che il giocatore MAX può ottenere in uno dei nodi antenati
- β è il punteggio massimo che il giocatore MIN può ottenere in uno dei nodi antenati

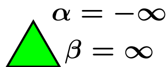
I valori α e β vengono aggiornati durante la ricerca, rispettivamente dai nodi "max" e dai nodi "min".

Nel momento in cui in un nodo $\alpha \geq \beta$ l'espansione di quel nodo viene interrotta.

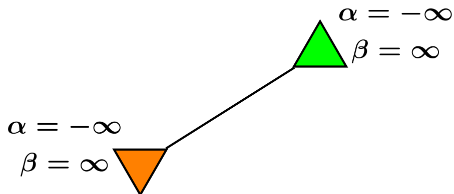
Alpha-Beta pruning

```
1  int alpha_beta(node_t node, int alpha, int beta, bool maxnode) {
2      if (/* nodo e' terminale */) {
3          return /* valore del nodo */;
4      }
5
6      int v;
7      if (maxnode) {
8          v = -INF;
9          for (node_t child : /* figli del nodo */) {
10             v = max(v, alpha_beta(child, alpha, beta, false));
11             alpha = max(v, alpha);
12             if (alpha >= beta) break;
13         }
14     } else {
15         v = INF;
16         for (node_t child : /* figli del nodo */) {
17             v = min(v, alpha_beta(child, alpha, beta, true));
18             beta = min(v, beta);
19             if (alpha >= beta) break;
20         }
21     }
22
23     return v;
24 }
```

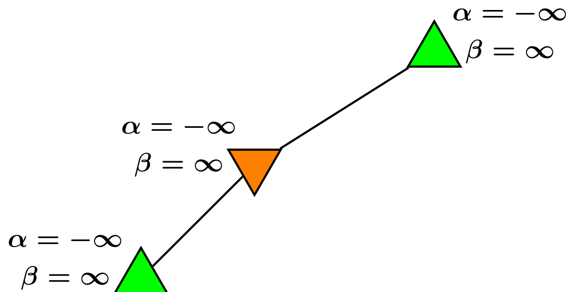
Alpha-Beta pruning



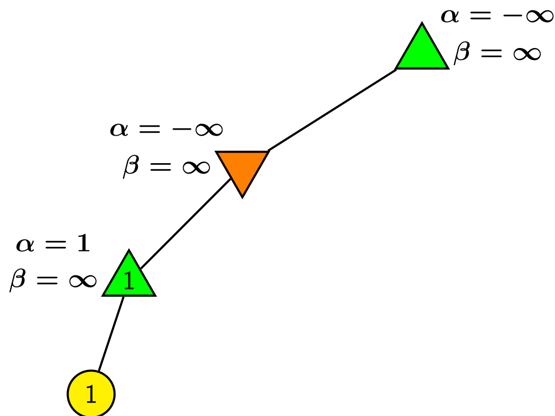
Alpha-Beta pruning



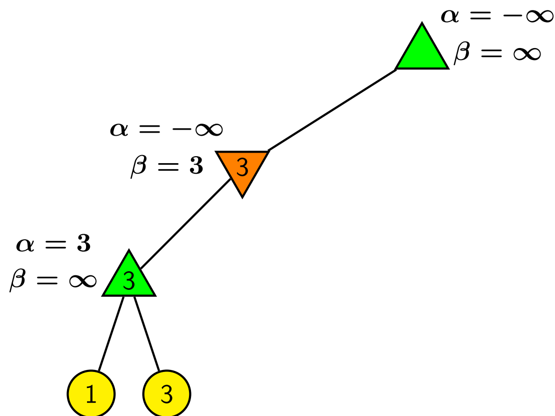
Alpha-Beta pruning



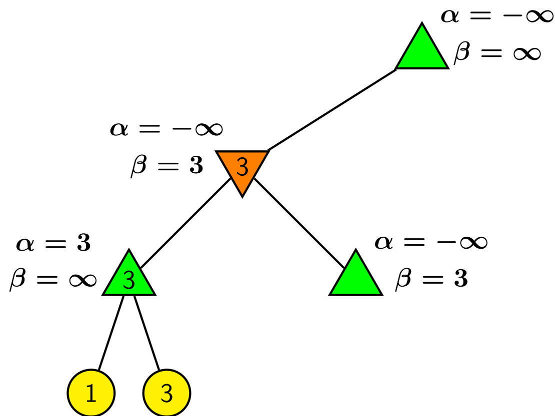
Alpha-Beta pruning



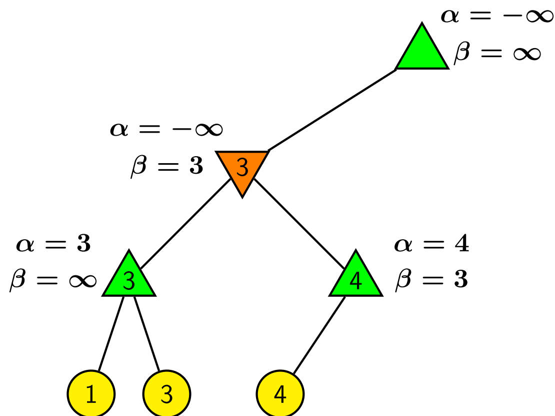
Alpha-Beta pruning



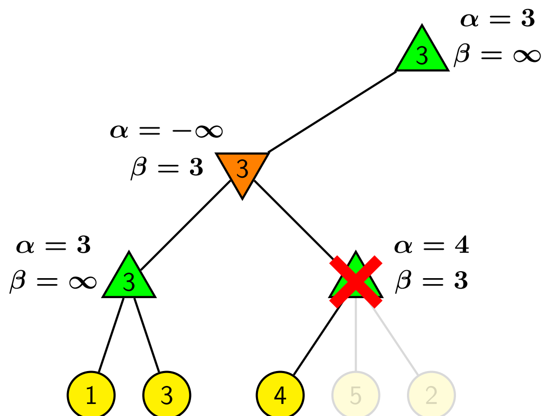
Alpha-Beta pruning



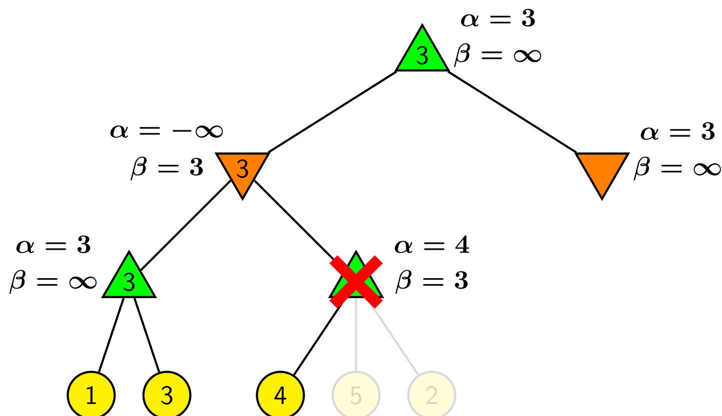
Alpha-Beta pruning



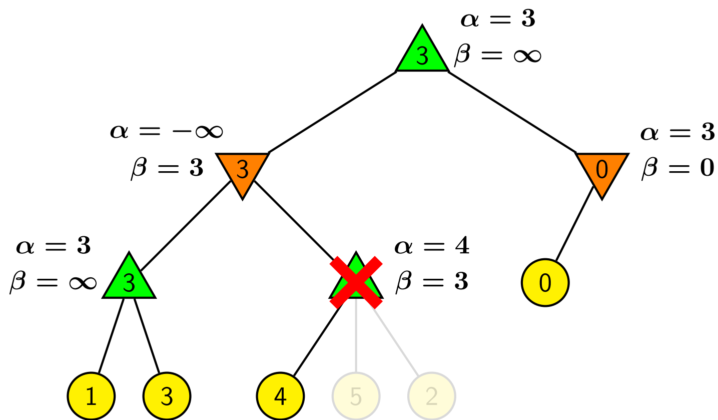
Alpha-Beta pruning



Alpha-Beta pruning



Alpha-Beta pruning



Alpha-Beta pruning