Obcinanie gałęzi w drzewach gier bezstronnych

dr inż. Piotr Beling

Uniwersytet Łódzki

2021

http://pbeling.w8.pl

Skończona, normalna gra bezstronna

Skończona, normalna gra bezstronna to trójka $(\mathbb{S}, N, i)$, taka że:

- $(\mathbb{S}, N)$ jest acyklicznym grafem skierowanym;
- $\mathbb{S}$ jest skończonym zbiorem pozycji (wierzchołków grafu);
- $N : \mathbb{S} \to 2^{\mathbb{S}}$ jest funkcją następników (definiującą krawędzie grafu);
  $N(p)$ jest zbiorem pozycji osiągalnych z p poprzez wykonanie jednego ruchu;
- $i \in \mathbb{S}$ jest pozycją początkową, od której rozpoczyna się gra.

Dwóch graczy wykonuje ruchy naprzemiennie.
Przegrywa gracz który nie może wykonać ruchu (jest tak gdy gra osiągnie pozycję $p \in \mathbb{S}$, taką że $N(p) = \emptyset$).

Skończona, normalna gra bezstronna

Skończona, normalna gra bezstronna to trójka $(\mathbb{S}, N, i)$, taka że:

- $(\mathbb{S}, N)$ jest acyklicznym grafem skierowanym;
- $\mathbb{S}$ jest skończonym zbiorem pozycji (wierzchołków grafu);
- $N : \mathbb{S} \to 2^{\mathbb{S}}$ jest funkcją następników (definiującą krawędzie grafu);
  $N(p)$ jest zbiorem pozycji osiągalnych z p poprzez wykonanie jednego ruchu;
- $i \in \mathbb{S}$ jest pozycją początkową, od której rozpoczyna się gra.

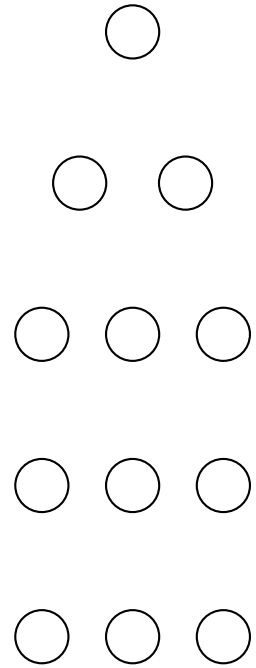
Dwóch graczy wykonuje ruchy naprzemiennie.
Przegrywa gracz który nie może wykonać ruchu (jest tak gdy gra osiągnie pozycję $p \in \mathbb{S}$, taką że $N(p) = \emptyset$).

Skończona, normalna gra bezstronna

Skończona, normalna gra bezstronna to trójka $(\mathbb{S}, N, i)$, taka że:

- $(\mathbb{S}, N)$ jest acyklicznym grafem skierowanym;
- $\mathbb{S}$ jest skończonym zbiorem pozycji (wierzchołków grafu);
- $N : \mathbb{S} \to 2^{\mathbb{S}}$ jest funkcją następników (definiującą krawędzie grafu);
  $N(p)$ jest zbiorem pozycji osiągalnych z p poprzez wykonanie jednego ruchu;
- $i \in \mathbb{S}$ jest pozycją początkową, od której rozpoczyna się gra.

Dwóch graczy wykonuje ruchy naprzemiennie.
Przegrywa gracz który nie może wykonać ruchu (jest tak gdy gra osiągnie pozycję $p \in \mathbb{S}$, taką że $N(p) = \emptyset$).

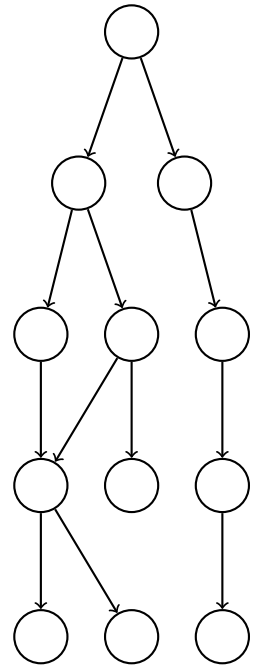


Skończona, normalna gra bezstronna

Skończona, normalna gra bezstronna to trójka $(\mathbb{S}, N, i)$, taka że:

- $(\mathbb{S}, N)$ jest acyklicznym grafem skierowanym;
- $\mathbb{S}$ jest skończonym zbiorem pozycji (wierzchołków grafu);
- $N : \mathbb{S} \to 2^{\mathbb{S}}$ jest funkcją następników (definiującą krawędzie grafu);
  $N(p)$ jest zbiorem pozycji osiągalnych z p poprzez wykonanie jednego ruchu;
- $i \in \mathbb{S}$ jest pozycją początkową, od której rozpoczyna się gra.

Dwóch graczy wykonuje ruchy naprzemiennie.
Przegrywa gracz który nie może wykonać ruchu (jest tak gdy gra osiągnie pozycję $p \in \mathbb{S}$, taką że $N(p) = \emptyset$).

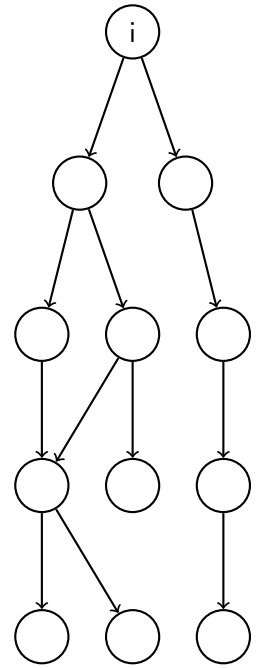


Skończona, normalna gra bezstronna

Skończona, normalna gra bezstronna to trójka $(\mathbb{S}, N, i)$, taka że:

- $(\mathbb{S}, N)$ jest acyklicznym grafem skierowanym;
- $\mathbb{S}$ jest skończonym zbiorem pozycji (wierzchołków grafu);
- $N : \mathbb{S} \to 2^{\mathbb{S}}$ jest funkcją następników (definiującą krawędzie grafu);
  $N(p)$ jest zbiorem pozycji osiągalnych z p poprzez wykonanie jednego ruchu;
- $i \in \mathbb{S}$ jest pozycją początkową, od której rozpoczyna się gra.

Dwóch graczy wykonuje ruchy naprzemiennie.
Przegrywa gracz który nie może wykonać ruchu (jest tak gdy gra osiągnie pozycję $p \in \mathbb{S}$, taką że $N(p) = \emptyset$).



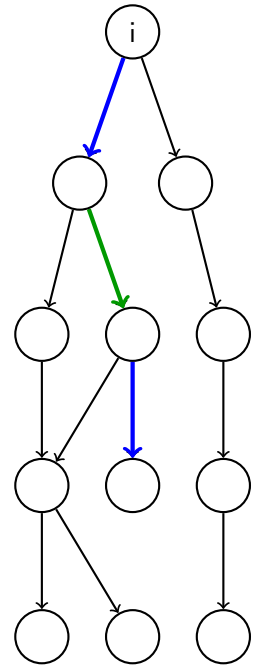
Skończona, normalna gra bezstronna

Skończona, normalna gra bezstronna to trójka $(\mathbb{S}, N, i)$, taka że:

- $(\mathbb{S}, N)$ jest acyklicznym grafem skierowanym;
- $\mathbb{S}$ jest skończonym zbiorem pozycji (wierzchołków grafu);
- $N : \mathbb{S} \to 2^{\mathbb{S}}$ jest funkcją następników (definiującą krawędzie grafu);
  $N(p)$ jest zbiorem pozycji osiągalnych z p poprzez wykonanie jednego ruchu;
- $i \in \mathbb{S}$ jest pozycją początkową, od której rozpoczyna się gra.

Dwóch graczy wykonuje ruchy naprzemiennie.

Przegrywa gracz który nie może wykonać ruchu (jest tak gdy gra osiągnie pozycję $p \in \mathbb{S}$, taką że $N(p) = \emptyset$).

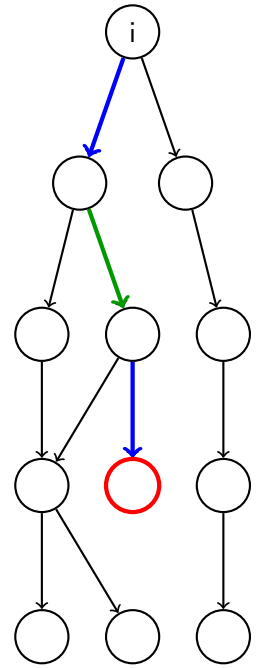


Skończona, normalna gra bezstronna

Skończona, normalna gra bezstronna to trójka $(\mathbb{S}, N, i)$, taka że:

- $(\mathbb{S}, N)$ jest acyklicznym grafem skierowanym;
- $\mathbb{S}$ jest skończonym zbiorem pozycji (wierzchołków grafu);
- $N : \mathbb{S} \to 2^{\mathbb{S}}$ jest funkcją następników (definiującą krawędzie grafu);
  $N(p)$ jest zbiorem pozycji osiągalnych z p poprzez wykonanie jednego ruchu;
- $i \in \mathbb{S}$ jest pozycją początkową, od której rozpoczyna się gra.

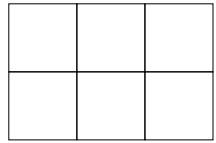
Dwóch graczy wykonuje ruchy naprzemiennie.
Przegrywa gracz który nie może wykonać ruchu (jest tak gdy gra osiągnie pozycję $p \in \mathbb{S}$, taką że $N(p) = \emptyset$).



Przykład: gra Cram

- Rozgrywka Crama toczy się prostokątnej planszy, podzielonej siatką na kwadratowe pola;
- Ruch polega na zajęciu dwóch dotychczas pustych, sąsiadujących ze sobą pól (w poziomie lub w pionie);
- przegrywa gracz który nie może wykonać ruchu.

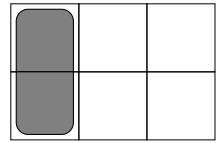
Przykładowy przebieg rozgrywki na planszy 3×2:



Przykład: gra Cram

- Rozgrywka Crama toczy się prostokątnej planszy, podzielonej siatką na kwadratowe pola;
- Ruch polega na zajęciu dwóch dotychczas pustych, sąsiadujących ze sobą pól (w poziomie lub w pionie);
- przegrywa gracz który nie może wykonać ruchu.

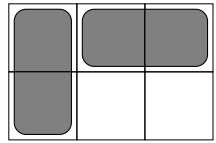
Przykładowy przebieg rozgrywki na planszy 3×2:



Przykład: gra Cram

- Rozgrywka Crama toczy się prostokątnej planszy, podzielonej siatką na kwadratowe pola;
- Ruch polega na zajęciu dwóch dotychczas pustych, sąsiadujących ze sobą pól (w poziomie lub w pionie);
- przegrywa gracz który nie może wykonać ruchu.

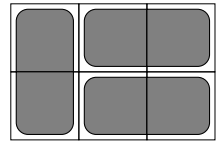
Przykładowy przebieg rozgrywki na planszy 3×2:



Przykład: gra Cram

- Rozgrywka Crama toczy się prostokątnej planszy, podzielonej siatką na kwadratowe pola;
- Ruch polega na zajęciu dwóch dotychczas pustych, sąsiadujących ze sobą pól (w poziomie lub w pionie);
- przegrywa gracz który nie może wykonać ruchu.

Przykładowy przebieg rozgrywki na planszy 3×2:



Definicja: funkcja mex

$\text{mex}(V)$ to najmniejsza liczba naturalna niezawarta w zbiorze V.
Na przykład: $\text{mex}\{0, 1, 3, 6\} = 2$, $\text{mex}\{1, 2, 3\} = 0$, $\text{mex}(\emptyset) = 0$.

Definicja: funkcja Sprague'a-Grundy'ego G, nimber

Dla dowolnej pozycji p, definiujemy *nimber* p, $G(p)$ jako:

$$G(p) = \text{mex}\{G(t) : t \in N(p)\}.$$

Uwaga: $G(p) = 0$ dla $N(p) = \emptyset$, co stanowi przypadek bazowy rekursji.

Definicja: funkcja mex

$\text{mex}(V)$ to najmniejsza liczba naturalna niezawarta w zbiorze V.
Na przykład: $\text{mex}\{0,1,3,6\} = 2$, $\text{mex}\{1,2,3\} = 0$, $\text{mex}(\emptyset) = 0$.

Definicja: funkcja Sprague'a-Grundy'ego G, nimber

Dla dowolnej pozycji p, definiujemy *nimber* p, $G(p)$ jako:

$$G(p) = \text{mex}\{G(t) : t \in N(p)\}.$$

Uwaga: $G(p) = 0$ dla $N(p) = \emptyset$, co stanowi przypadek bazowy rekursji.

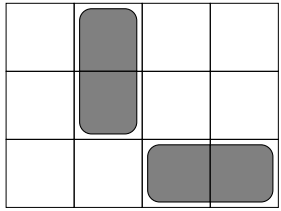
Twierdzenie Sprague'a-Grundy'ego

- Dowolna pozycja p jest wygrana (dla gracza który wykonuje w niej ruch) wtedy, i tylko wtedy gdy $G(p) \neq 0$;
- jeśli p składa się z rozłącznych pozycji $p_1, p_2, \ldots, p_k$ (by wykonać ruch w p, gracz wybiera jedną z pozycji $p_1, p_2, \ldots, p_k$ i wykonuje w niej ruch) wtedy

$$G(p) = G(p_1) \oplus G(p_2) \oplus \cdots \oplus G(p_k),$$

gdzie $\oplus$ to bitowa operacja xor, zwana też: sumą modulo 2 bez przeniesień, bitową alternatywą wykluczającą, bitowym „albo", nim-sumą.

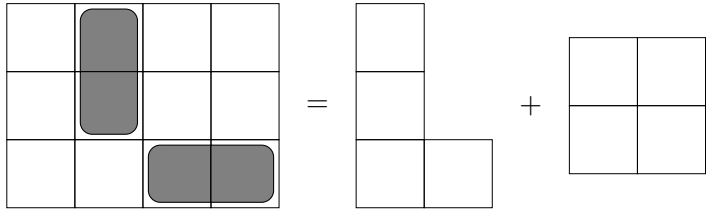
Przykład dekompozycji pozycji Crama



p

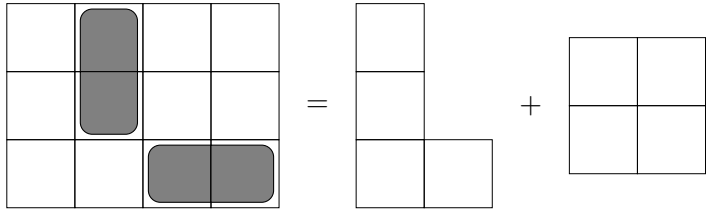
- **Przedstawiona pozycja Crama** jest sumą dwóch składowych (ruch można wykonać jedynie w wybranej składowej, bez wpływu na pozostałe).
- możemy niezależnie (np. z definicji) wyznaczyć nimbery: $G(a) = 2$, $G(b) = 0$,
- i z twierdzenia Sprague'a-Grundy'ego: $G(p) = G(a) \oplus G(b) = 2 \oplus 0 = 2$.
- Ponieważ $G(p) \neq 0$, p jest wygrany (dla gracza do którego należy ruch).

Przykład dekompozycji pozycji Crama



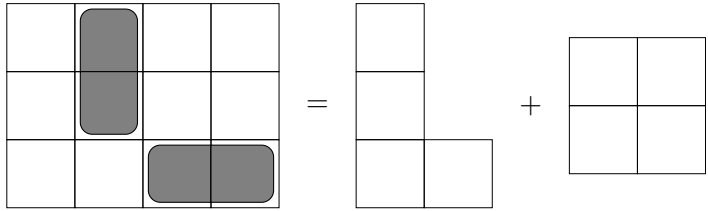
- Przedstawiona pozycja Crama jest sumą dwóch składowych (ruch można wykonać jedynie w wybranej składowej, bez wpływu na pozostałe).
- możemy niezależnie (np. z definicji) wyznaczyć nimbery: $G(a) = 2$, $G(b) = 0$,
- i z twierdzenia Sprague'a-Grundy'ego: $G(p) = G(a) \oplus G(b) = 2 \oplus 0 = 2$.
- Ponieważ $G(p) \neq 0$, p jest wygrany (dla gracza do którego należy ruch).

Przykład dekompozycji pozycji Crama



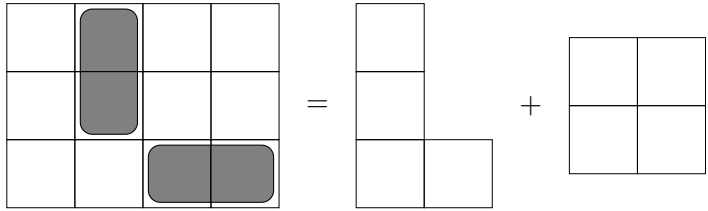
- Przedstawiona pozycja Crama jest sumą dwóch składowych (ruch można wykonać jedynie w wybranej składowej, bez wpływu na pozostałe).
- możemy niezależnie (np. z definicji) wyznaczyć nimbery: $G(a) = 2$, $G(b) = 0$,
- i z twierdzenia Sprague'a-Grundy'ego: $G(p) = G(a) \oplus G(b) = 2 \oplus 0 = 2$.
- Ponieważ $G(p) \neq 0$, p jest wygrany (dla gracza do którego należy ruch).

Przykład dekompozycji pozycji Crama



- Przedstawiona pozycja Crama jest sumą dwóch składowych (ruch można wykonać jedynie w wybranej składowej, bez wpływu na pozostałe).
- możemy niezależnie (np. z definicji) wyznaczyć nimbery: $G(a) = 2$, $G(b) = 0$,
- i z twierdzenia Sprague'a-Grundy'ego: $G(p) = G(a) \oplus G(b) = 2 \oplus 0 = 2$.
- Ponieważ $G(p) \neq 0$, p jest wygrany (dla gracza do którego należy ruch).

Przykład dekompozycji pozycji Crama

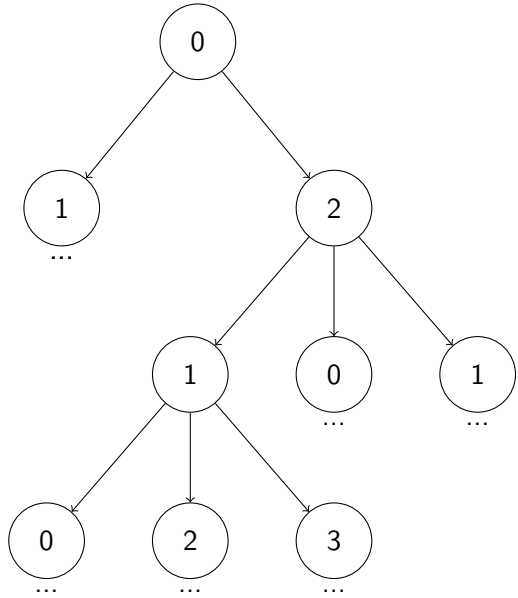


- Przedstawiona pozycja Crama jest sumą dwóch składowych (ruch można wykonać jedynie w wybranej składowej, bez wpływu na pozostałe).
- możemy niezależnie (np. z definicji) wyznaczyć nimbery: $G(a) = 2$, $G(b) = 0$,
- i z twierdzenia Sprague'a-Grundy'ego: $G(p) = G(a) \oplus G(b) = 2 \oplus 0 = 2$.
- Ponieważ $G(p) \neq 0$, p jest wygrany (dla gracza do którego należy ruch).

Obliczanie nimbera s z definicji

```
1   fun def(s):
2       P ← {0, 1, ..., |N(s)|}
3       for m ∈ N(s):
4           v ← def(m)
5           if v ∈ P:
6               P ← P \ {v}
7           else:
8               P ← P \ {max(P)}
9       result ← jedyny element w P
10      return result
```

- P to zbiór potencjalnych nimberów s.
- Algorytm wyrzuca z P po jednym elemencie po zbadaniu każdego z następników pozycji s.
- P ma $|N(s)| + 1$ elementów na początku i 1 element (nimber pozycji s) na końcu.

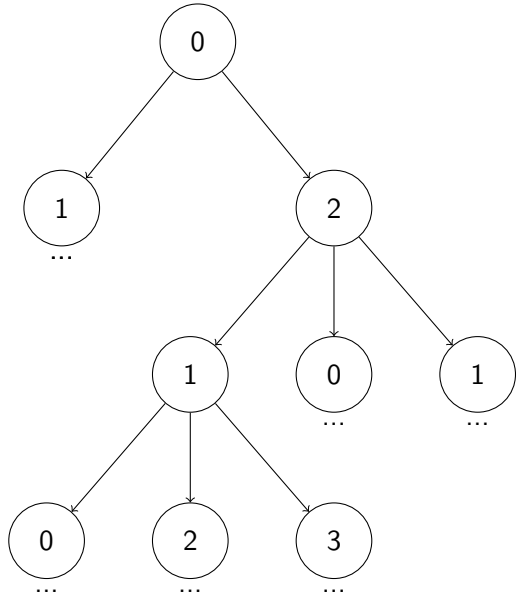


Przykładowy fragment drzewa poszukiwań.

Obliczanie nimbera s z definicji

```
1   fun def(s):
2       P ← {0, 1, ..., |N(s)|}
3       for m ∈ N(s):
4           v ← def(m)
5           if v ∈ P:
6               P ← P \ {v}
7           else:
8               P ← P \ {max(P)}
9       result ← jedyny element w P
10      return result
```

- P to zbiór potencjalnych nimberów s.
- Algorytm wyrzuca z P po jednym elemencie po zbadaniu każdego z następników pozycji s.
- P ma $|N(s)| + 1$ elementów na początku i 1 element (nimber pozycji s) na końcu.

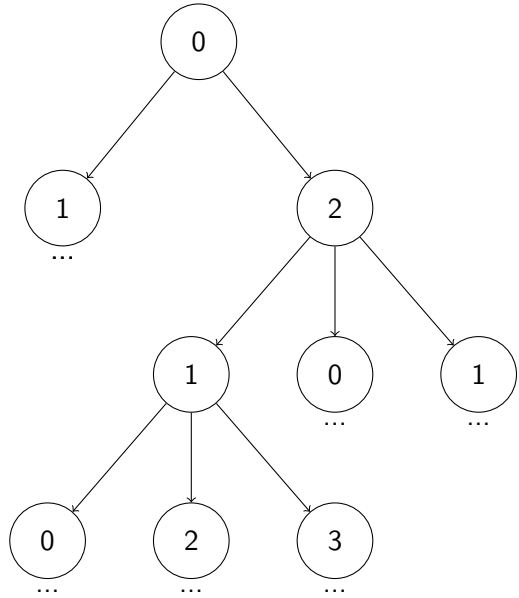


Przykładowy fragment drzewa poszukiwań.

Obliczanie nimbera s z definicji

```
1   fun def(s):
2       P ← {0, 1, ..., |N(s)|}
3       for m ∈ N(s):
4           v ← def(m)
5           if v ∈ P:
6               P ← P \ {v}
7           else:
8               P ← P \ {max(P)}
9       result ← jedyny element w P
10      return result
```

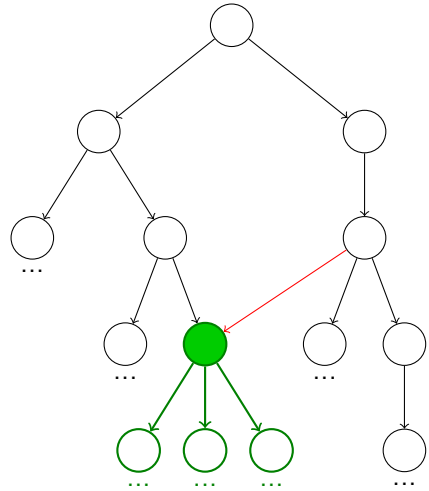
- P to zbiór potencjalnych nimberów s.
- Algorytm wyrzuca z P po jednym elemencie po zbadaniu każdego z następników pozycji s.
- P ma $|N(s)| + 1$ elementów na początku i 1 element (nimber pozycji s) na końcu.



Przykładowy fragment drzewa poszukiwań.

Obliczanie nimbera s z definicji + Tablica Transpozycji

```
1   fun def(s):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           v ← def(m)
6           if v ∈ P:
7               P ← P \ {v}
8           else:
9               P ← P \ {max(P)}
10      result ← jedyny element w P
11      TT[s] ← result
12      return result
```



Tablica transpozycji (TT) przechowuje nimbery dotychczas zbadanych pozycji. Pozwala ona uniknąć redundatnych obliczeń, gdy ta sama pozycja może być osiągnięta po wielu różnych sekwencjach ruchów.

Spostrzeżenie:

```
1   fun def(s):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           v ← def(m)
6           if v ∈ P:
7               P ← P \ {v}
8           else:
9               P ← P \ {max(P)}
10      result ← jedyny element w P
11      TT[s] ← result
12      return result
```

- Gdy $v \notin P$, algorytm wyklucza $\max(P)$ z P w linii 9, niezależnie od dokładnej wartości v.
- To samo robi w linii 7 gdy $v = \max(P)$.
- Więc znajomość dokładnej wartości v jest niezbędna tylko gdy $v \in P \setminus \max(P)$ (by wykluczyć v z P w linii 7).
- W pozostałych wypadkach (by wykluczyć $\max(P)$ z P) wystarczy wiedzieć, że $v \notin P \setminus \max(P)$.

Spostrzeżenie:

```
1   fun def(s):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           v ← def(m)
6           if v ∈ P:
7               P ← P \ {v}
8           else:
9               P ← P \ {max(P)}
10      result ← jedyny element w P
11      TT[s] ← result
12      return result
```

- Gdy $v \notin P$, algorytm wyklucza $\max(P)$ z P w linii 9, niezależnie od dokładnej wartości v.
- To samo robi w linii 7 gdy $v = \max(P)$.
- Więc znajomość dokładnej wartości v jest niezbędna tylko gdy $v \in P \setminus \max(P)$ (by wykluczyć v z P w linii 7).
- W pozostałych wypadkach (by wykluczyć $\max(P)$ z P) wystarczy wiedzieć, że $v \notin P \setminus \max(P)$.

Spostrzeżenie:

```
1   fun def(s):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           v ← def(m)
6           if v ∈ P:
7               P ← P \ {v}
8           else:
9               P ← P \ {max(P)}
10      result ← jedyny element w P
11      TT[s] ← result
12      return result
```

- Gdy $v \notin P$, algorytm wyklucza $\max(P)$ z P w linii 9, niezależnie od dokładnej wartości v.
- To samo robi w linii 7 gdy $v = \max(P)$.
- Więc znajomość dokładnej wartości v jest niezbędna tylko gdy $v \in P \setminus \max(P)$ (by wykluczyć v z P w linii 7).
- W pozostałych wypadkach (by wykluczyć $\max(P)$ z P) wystarczy wiedzieć, że $v \notin P \setminus \max(P)$.

Spostrzeżenie:

```
1   fun def(s):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           v ← def(m)
6           if v ∈ P:
7               P ← P \ {v}
8           else:
9               P ← P \ {max(P)}
10      result ← jedyny element w P
11      TT[s] ← result
12      return result
```

- Gdy $v \notin P$, algorytm wyklucza $\max(P)$ z P w linii 9, niezależnie od dokładnej wartości v.
- To samo robi w linii 7 gdy $v = \max(P)$.
- Więc znajomość dokładnej wartości v jest niezbędna tylko gdy $v \in P \setminus \max(P)$ (by wykluczyć v z P w linii 7).
- W pozostałych wypadkach (by wykluczyć $\max(P)$ z P) wystarczy wiedzieć, że $v \notin P \setminus \max(P)$.

Uproszczone odcięcia + Tablica Transpozycji

```
1   fun scut(s, R):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           if P ∩ R = ∅: return −1
6           v ← scut(m, P \ {max(P)})
7           if v ∈ P:
8               P ← P \ {v}
9           else:
10              P ← P \ {max(P)}
11      result ← jedyny element w P
12      TT[s] ← result
13      return result
```

- Zbiór $P \setminus \max(P)$ jest przekazywany do rekurencyjnego wywołania w linii 6.
- To wywołanie jest zobligowane zwrócić dokładnego nimbera tylko gdy jest on zawarty w tym zbiorze (oznaczonym w wywołaniu przez R).
- W przeciwnym razie, może ono zwrócić albo dokładną wartość albo specjalną wartość -1 (niezawartą w żadnym zbiorze nimberów).
- Algorytm wykorzystuje argument R do wcześniejszego zakończenia obliczeń (odcięcia) w linii 5,
- gdy tylko jest on w stanie dowieść, że $G(s) \notin R$, bo zbiór potencjalnych wartości s jest rozłączny z R.
- Zwraca wtedy -1.

Uproszczone odcięcia + Tablica Transpozycji

```
1   fun scut(s, R):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           if P ∩ R = ∅: return −1
6           v ← scut(m, P \ {max(P)})
7           if v ∈ P:
8               P ← P \ {v}
9           else:
10              P ← P \ {max(P)}
11      result ← jedyny element w P
12      TT[s] ← result
13      return result
```

- Zbiór $P \setminus \max(P)$ jest przekazywany do rekurencyjnego wywołania w linii 6.
- To wywołanie jest zobligowane zwrócić dokładnego nimbera tylko gdy jest on zawarty w tym zbiorze (oznaczonym w wywołaniu przez R).
- W przeciwnym razie, może ono zwrócić albo dokładną wartość albo specjalną wartość -1 (niezawartą w żadnym zbiorze nimberów).
- Algorytm wykorzystuje argument R do wcześniejszego zakończenia obliczeń (odcięcia) w linii 5,
- gdy tylko jest on w stanie dowieść, że $G(s) \notin R$, bo zbiór potencjalnych wartości s jest rozłączny z R.
- Zwraca wtedy -1.

Uproszczone odcięcia + Tablica Transpozycji

```
1   fun scut(s, R):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           if P ∩ R = ∅: return −1
6           v ← scut(m, P \ {max(P)})
7           if v ∈ P:
8               P ← P \ {v}
9           else:
10              P ← P \ {max(P)}
11      result ← jedyny element w P
12      TT[s] ← result
13      return result
```

- Zbiór $P \setminus \max(P)$ jest przekazywany do rekurencyjnego wywołania w linii 6.
- To wywołanie jest zobligowane zwrócić dokładnego nimbera tylko gdy jest on zawarty w tym zbiorze (oznaczonym w wywołaniu przez R).
- W przeciwnym razie, może ono zwrócić albo dokładną wartość albo specjalną wartość -1 (niezawartą w żadnym zbiorze nimberów).
- Algorytm wykorzystuje argument R do wcześniejszego zakończenia obliczeń (odcięcia) w linii 5,
- gdy tylko jest on w stanie dowieść, że $G(s) \notin R$, bo zbiór potencjalnych wartości s jest rozłączny z R.
- Zwraca wtedy -1.

Uproszczone odcięcia + Tablica Transpozycji

```
1   fun scut(s, R):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           if P ∩ R = ∅: return −1
6           v ← scut(m, P \ {max(P)})
7           if v ∈ P:
8               P ← P \ {v}
9           else:
10              P ← P \ {max(P)}
11      result ← jedyny element w P
12      TT[s] ← result
13      return result
```

- Zbiór $P \setminus \max(P)$ jest przekazywany do rekurencyjnego wywołania w linii 6.
- To wywołanie jest zobligowane zwrócić dokładnego nimbera tylko gdy jest on zawarty w tym zbiorze (oznaczonym w wywołaniu przez R).
- W przeciwnym razie, może ono zwrócić albo dokładną wartość albo specjalną wartość -1 (niezawartą w żadnym zbiorze nimberów).
- Algorytm wykorzystuje argument R do wcześniejszego zakończenia obliczeń (odcięcia) w linii 5,
- gdy tylko jest on w stanie dowieść, że $G(s) \notin R$, bo zbiór potencjalnych wartości s jest rozłączny z R.
- Zwraca wtedy -1.

Uproszczone odcięcia + Tablica Transpozycji

```
1   fun scut(s, R):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           if P ∩ R = ∅: return −1
6           v ← scut(m, P \ {max(P)})
7           if v ∈ P:
8               P ← P \ {v}
9           else:
10              P ← P \ {max(P)}
11      result ← jedyny element w P
12      TT[s] ← result
13      return result
```

- Zbiór $P \setminus \max(P)$ jest przekazywany do rekurencyjnego wywołania w linii 6.
- To wywołanie jest zobligowane zwrócić dokładnego nimbera tylko gdy jest on zawarty w tym zbiorze (oznaczonym w wywołaniu przez R).
- W przeciwnym razie, może ono zwrócić albo dokładną wartość albo specjalną wartość -1 (niezawartą w żadnym zbiorze nimberów).
- Algorytm wykorzystuje argument R do wcześniejszego zakończenia obliczeń (odcięcia) w linii 5,
- gdy tylko jest on w stanie dowieść, że $G(s) \notin R$, bo zbiór potencjalnych wartości s jest rozłączny z R.
- Zwraca wtedy -1.

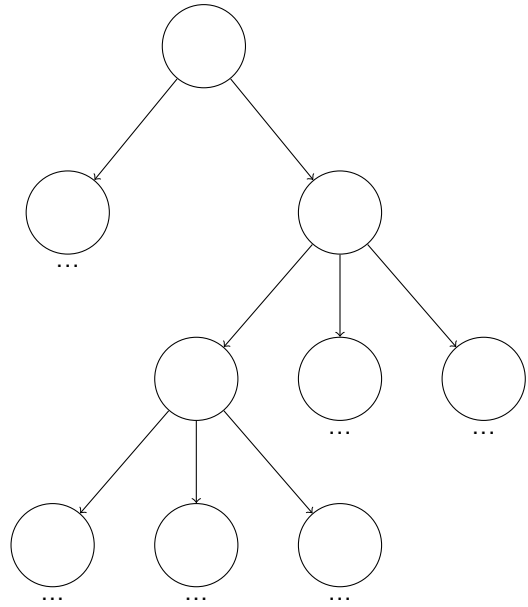
Uproszczone odcięcia + Tablica Transpozycji

```
1   fun scut(s, R):
2       if s ∈ TT: return TT[s]
3       P ← {0, 1, ..., |N(s)|}
4       for m ∈ N(s):
5           if P ∩ R = ∅: return −1
6           v ← scut(m, P \ {max(P)})
7           if v ∈ P:
8               P ← P \ {v}
9           else:
10              P ← P \ {max(P)}
11      result ← jedyny element w P
12      TT[s] ← result
13      return result
```

- Zbiór $P \setminus \max(P)$ jest przekazywany do rekurencyjnego wywołania w linii 6.
- To wywołanie jest zobligowane zwrócić dokładnego nimbera tylko gdy jest on zawarty w tym zbiorze (oznaczonym w wywołaniu przez R).
- W przeciwnym razie, może ono zwrócić albo dokładną wartość albo specjalną wartość -1 (niezawartą w żadnym zbiorze nimberów).
- Algorytm wykorzystuje argument R do wcześniejszego zakończenia obliczeń (odcięcia) w linii 5,
- gdy tylko jest on w stanie dowieść, że $G(s) \notin R$, bo zbiór potencjalnych wartości s jest rozłączny z R.
- Zwraca wtedy -1.

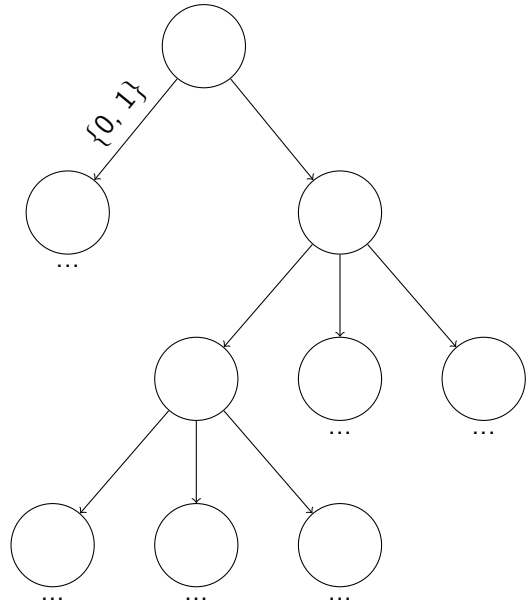
Uproszczone odcięcia – przykładowy fragment drzewa poszukiwań

- Ponieważ korzeń ma 2 następniki, to początkową wartością jego zbioru P jest $\{0, 1, 2\}$.
- $P \setminus \max(P) = \{0, 1\}$ jest przekazany do pierwszego z wywołań.
- Ono zwraca 1, i ponieważ $1 \in \{0, 1, 2\}$, to jest ona usuwana z P (teraz $P = \{0, 2\}$).



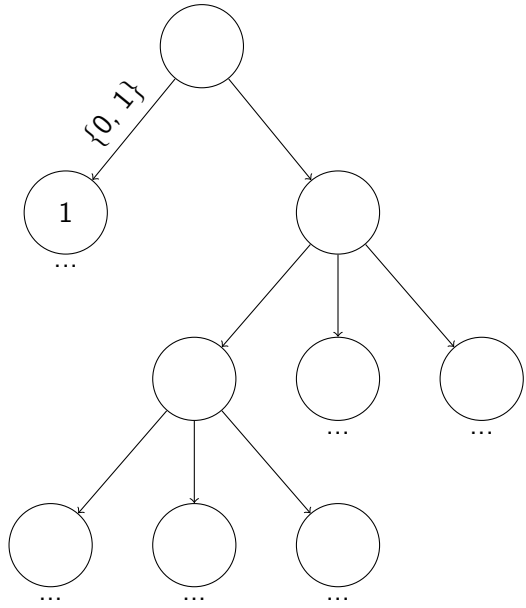
Uproszczone odcięcia – przykładowy fragment drzewa poszukiwań

- Ponieważ korzeń ma 2 następniki, to początkową wartością jego zbioru P jest $\{0, 1, 2\}$.
- $P \setminus \max(P) = \{0, 1\}$ jest przekazany do pierwszego z wywołań.
- Ono zwraca 1, i ponieważ $1 \in \{0, 1, 2\}$, to jest ona usuwana z P (teraz $P = \{0, 2\}$).



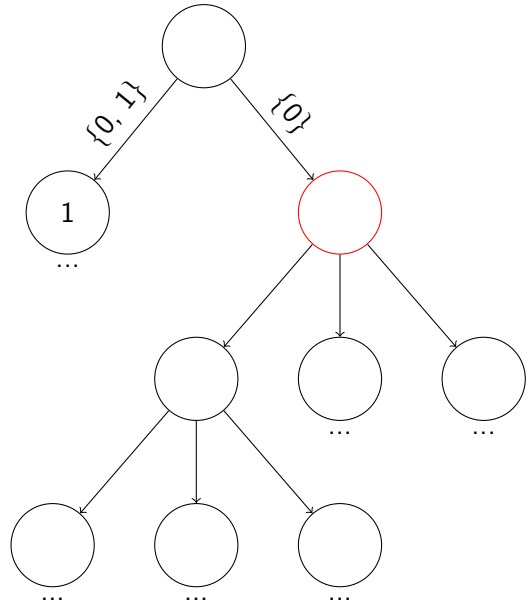
Uproszczone odcięcia – przykładowy fragment drzewa poszukiwań

- Ponieważ korzeń ma 2 następniki, to początkową wartością jego zbioru P jest $\{0, 1, 2\}$.
- $P \setminus \max(P) = \{0, 1\}$ jest przekazany do pierwszego z wywołań.
- Ono zwraca 1, i ponieważ $1 \in \{0, 1, 2\}$, to jest ona usuwana z P (teraz $P = \{0, 2\}$).



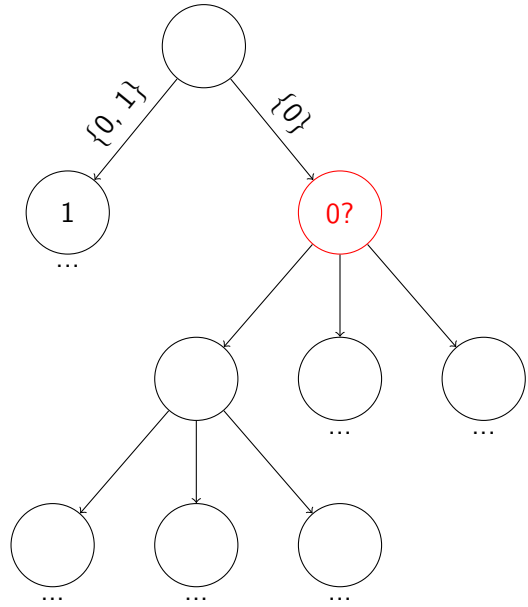
Uproszczone odcięcia – przykładowy fragment drzewa poszukiwań

- $P \setminus \max(P) = \{0\}$ jest przekazany do prawego syna.
- Teraz, by ustalić wartość korzenia, wystarczy wiedzieć czy nimber jego prawego syna wynosi 0 czy nie.
- Ten syn ma trzy następniki. Jeśli którykolwiek z nich ma nimber 0, to pozostałe można odciąć, ponieważ nimber prawego syna korzenia nie jest równy 0 (nie ma go w zbiorze $\{0\}$ przekazanym z korzenia).



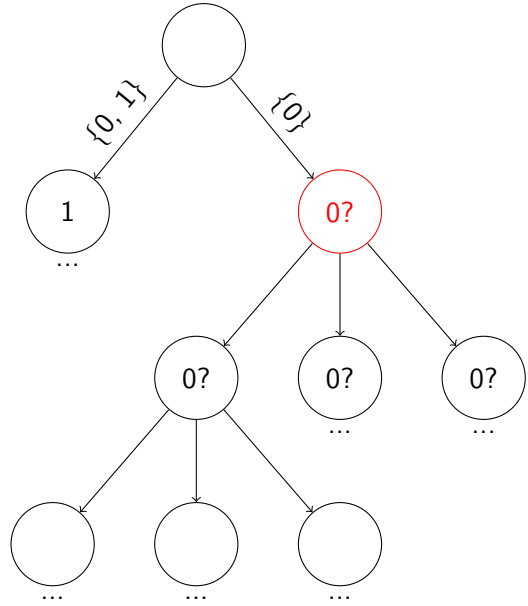
Uproszczone odcięcia – przykładowy fragment drzewa poszukiwań

- $P \setminus \max(P) = \{0\}$ jest przekazany do prawego syna.
- Teraz, by ustalić wartość korzenia, wystarczy wiedzieć czy nimber jego prawego syna wynosi 0 czy nie.
- Ten syn ma trzy następniki. Jeśli którykolwiek z nich ma nimber 0, to pozostałe można odciąć, ponieważ nimber prawego syna korzenia nie jest równy 0 (nie ma go w zbiorze $\{0\}$ przekazanym z korzenia).



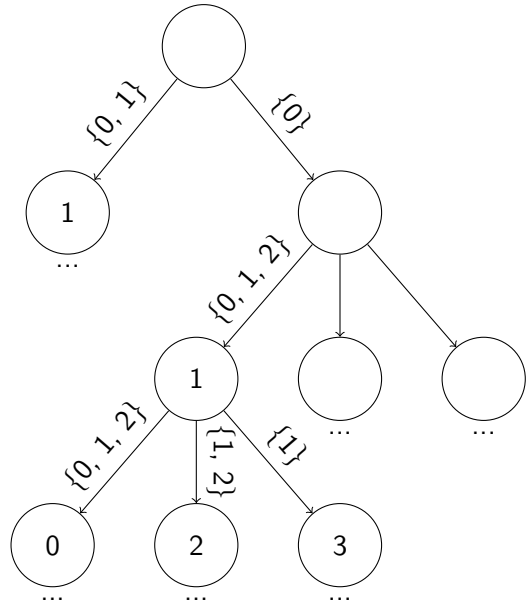
Uproszczone odcięcia – przykładowy fragment drzewa poszukiwań

- $P \setminus \max(P) = \{0\}$ jest przekazany do prawego syna.
- Teraz, by ustalić wartość korzenia, wystarczy wiedzieć czy nimber jego prawego syna wynosi 0 czy nie.
- Ten syn ma trzy następniki. Jeśli którykolwiek z nich ma nimber 0, to pozostałe można odciąć, ponieważ nimber prawego syna korzenia nie jest równy 0 (nie ma go w zbiorze $\{0\}$ przekazanym z korzenia).



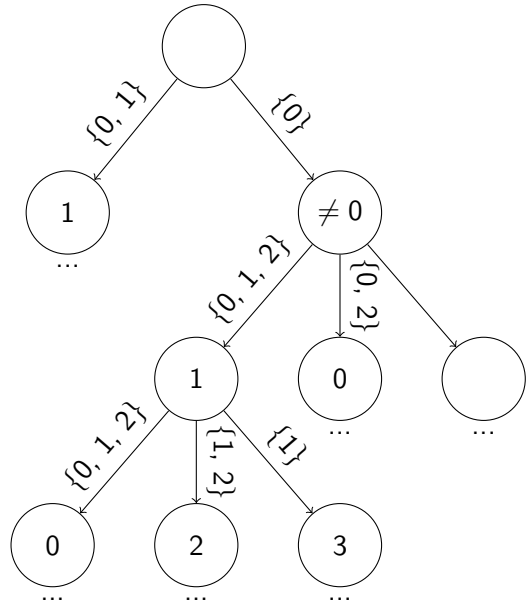
Uproszczone odcięcia – przykładowy fragment drzewa poszukiwań

- Pierwszy następnik ma nimber 1, co zostało stwierdzone po odwiedzeniu jego wszystkich następników.
- Drugi następnik ma nimber 0,
- więc trzeci może zostać odcięty.



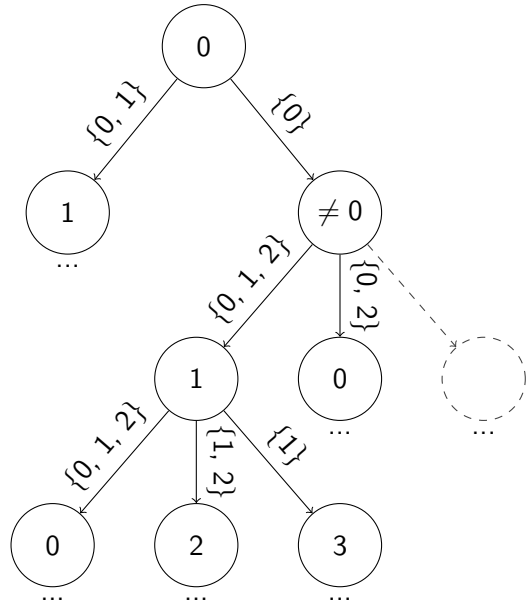
Uproszczone odcięcia – przykładowy fragment drzewa poszukiwań

- Pierwszy następnik ma nimber 1, co zostało stwierdzone po odwiedzeniu jego wszystkich następników.
- Drugi następnik ma nimber 0,
- więc trzeci może zostać odcięty.



Uproszczone odcięcia – przykładowy fragment drzewa poszukiwań

- Pierwszy następnik ma nimber 1, co zostało stwierdzone po odwiedzeniu jego wszystkich następników.
- Drugi następnik ma nimber 0,
- więc trzeci może zostać odcięty.



```
1  fun cut(s, R):
2      if s ∈ TT: return TT[s]
3      P ← {0, 1, ..., |N(s)|}
4      exact ← true
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← cut(m, (P \ {max(P)}) ∩
                  {0, ..., max(R)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12             if v = −1: exact ← false
13     result ← jedyny element w P
14     if exact or result ⩽ max(R):
15         TT[s] ← result
16         return result
17     else:
18         return −1
```

- Spostrzeżenie: Wywołujący nigdy nie żąda nimbera s gdy jest on większy od $\max(R)$.
- Dlatego zbiór przekazywany do wywołania rekurencyjnego (w linii 7) może zostać ograniczony do (tj. przecięty z) $\{0, 1, \ldots, \max(R)\}$.
- To może doprowadzić do:
- zwiększenia liczby odcięć w poddrzewie,
- ale także do niedokładności w P.

```
1  fun cut(s, R):
2      if s ∈ TT: return TT[s]
3      P ← {0, 1, ..., |N(s)|}
4      exact ← true
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← cut(m, (P \ {max(P)}) ∩
               {0, ..., max(R)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12             if v = −1: exact ← false
13     result ← jedyny element w P
14     if exact or result ⩽ max(R):
15         TT[s] ← result
16         return result
17     else:
18         return −1
```

- Spostrzeżenie: Wywołujący nigdy nie żąda nimbera s gdy jest on większy od $\max(R)$.
- Dlatego zbiór przekazywany do wywołania rekurencyjnego (w linii 7) może zostać ograniczony do (tj. przecięty z) $\{0, 1, \ldots, \max(R)\}$.
- To może doprowadzić do:
- zwiększenia liczby odcięć w poddrzewie,
- ale także do niedokładności w P.

```
1  fun cut(s, R):
2      if s ∈ TT: return TT[s]
3      P ← {0, 1, ..., |N(s)|}
4      exact ← true
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← cut(m, (P \ {max(P)}) ∩
               {0, ..., max(R)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12             if v = −1: exact ← false
13     result ← jedyny element w P
14     if exact or result ⩽ max(R):
15         TT[s] ← result
16         return result
17     else:
18         return −1
```

- Spostrzeżenie: Wywołujący nigdy nie żąda nimbera s gdy jest on większy od $\max(R)$.
- Dlatego zbiór przekazywany do wywołania rekurencyjnego (w linii 7) może zostać ograniczony do (tj. przecięty z) $\{0, 1, \ldots, \max(R)\}$.
- To może doprowadzić do:
- zwiększenia liczby odcięć w poddrzewie,
- ale także do niedokładności w P.

```
1  fun cut(s, R):
2      if s ∈ TT: return TT[s]
3      P ← {0, 1, ..., |N(s)|}
4      exact ← true
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← cut(m, (P \ {max(P)}) ∩
                  {0, ..., max(R)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12             if v = −1: exact ← false
13     result ← jedyny element w P
14     if exact or result ⩽ max(R):
15         TT[s] ← result
16         return result
17     else:
18         return −1
```

- Spostrzeżenie: Wywołujący nigdy nie żąda nimbera s gdy jest on większy od $\max(R)$.
- Dlatego zbiór przekazywany do wywołania rekurencyjnego (w linii 7) może zostać ograniczony do (tj. przecięty z) $\{0, 1, \ldots, \max(R)\}$.
- To może doprowadzić do:
- zwiększenia liczby odcięć w poddrzewie,
- ale także do niedokładności w P.

```
1  fun cut(s, R):
2      if s ∈ TT: return TT[s]
3      P ← {0, 1, ..., |N(s)|}
4      exact ← true
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← cut(m, (P \ {max(P)}) ∩
                   {0, ..., max(R)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12             if v = −1: exact ← false
13     result ← jedyny element w P
14     if exact or result ⩽ max(R):
15         TT[s] ← result
16         return result
17     else:
18         return −1
```

- Spostrzeżenie: Wywołujący nigdy nie żąda nimbera s gdy jest on większy od $\max(R)$.
- Dlatego zbiór przekazywany do wywołania rekurencyjnego (w linii 7) może zostać ograniczony do (tj. przecięty z) $\{0, 1, \ldots, \max(R)\}$.
- To może doprowadzić do:
- zwiększenia liczby odcięć w poddrzewie,
- ale także do niedokładności w P.

```
1  fun cut(s, R):
2      if s ∈ TT: return TT[s]
3      P ← {0, 1, ..., |N(s)|}
4      exact ← true
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← cut(m, (P \ {max(P)}) ∩
               {0, ..., max(R)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12             if v = −1: exact ← false
13     result ← jedyny element w P
14     if exact or result ⩽ max(R):
15         TT[s] ← result
16         return result
17     else:
18         return −1
```

- Jednakże, P jest poprawny:
- gdy wszystkie wywołania rekurencyjne zwróciły dokładne wartości (tj. żadne nie zwróciło -1); wtedy flaga exact pozostanie ustawiona na true;
- w zakresie wartości nieprzekraczających $\max(R)$.
- Tylko gdy powyższe warunki są spełnione (w linii 14), result zostanie zapisany do TT (w linii 15) i zwrócony.
- W przeciwnym razie zostanie zwrócone -1 w linii 18. Wtedy s nie zawiera się w R, bo jest większe od $\max(R)$.

```
1  fun cut(s, R):
2      if s ∈ TT: return TT[s]
3      P ← {0, 1, ..., |N(s)|}
4      exact ← true
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← cut(m, (P \ {max(P)}) ∩
                  {0, ..., max(R)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12             if v = −1: exact ← false
13     result ← jedyny element w P
14     if exact or result ⩽ max(R):
15         TT[s] ← result
16         return result
17     else:
18         return −1
```

- Jednakże, P jest poprawny:
- gdy wszystkie wywołania rekurencyjne zwróciły dokładne wartości (tj. żadne nie zwróciło -1); wtedy flaga exact pozostanie ustawiona na true;
- w zakresie wartości nieprzekraczających $\max(R)$.
- Tylko gdy powyższe warunki są spełnione (w linii 14), result zostanie zapisany do TT (w linii 15) i zwrócony.
- W przeciwnym razie zostanie zwrócone -1 w linii 18. Wtedy s nie zawiera się w R, bo jest większe od $\max(R)$.

```
1  fun cut(s, R):
2      if s ∈ TT: return TT[s]
3      P ← {0, 1, ..., |N(s)|}
4      exact ← true
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← cut(m, (P \ {max(P)}) ∩
                  {0, ..., max(R)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12             if v = −1: exact ← false
13     result ← jedyny element w P
14     if exact or result ⩽ max(R):
15         TT[s] ← result
16         return result
17     else:
18         return −1
```

- Jednakże, P jest poprawny:
- gdy wszystkie wywołania rekurencyjne zwróciły dokładne wartości (tj. żadne nie zwróciło -1); wtedy flaga exact pozostanie ustawiona na true;
- w zakresie wartości nieprzekraczających $\max(R)$.
- Tylko gdy powyższe warunki są spełnione (w linii 14), result zostanie zapisany do TT (w linii 15) i zwrócony.
- W przeciwnym razie zostanie zwrócone -1 w linii 18. Wtedy s nie zawiera się w R, bo jest większe od $\max(R)$.

```
1  fun cut(s, R):
2      if s ∈ TT: return TT[s]
3      P ← {0, 1, ..., |N(s)|}
4      exact ← true
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← cut(m, (P \ {max(P)}) ∩
                  {0, ..., max(R)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12             if v = −1: exact ← false
13     result ← jedyny element w P
14     if exact or result ⩽ max(R):
15         TT[s] ← result
16         return result
17     else:
18         return −1
```

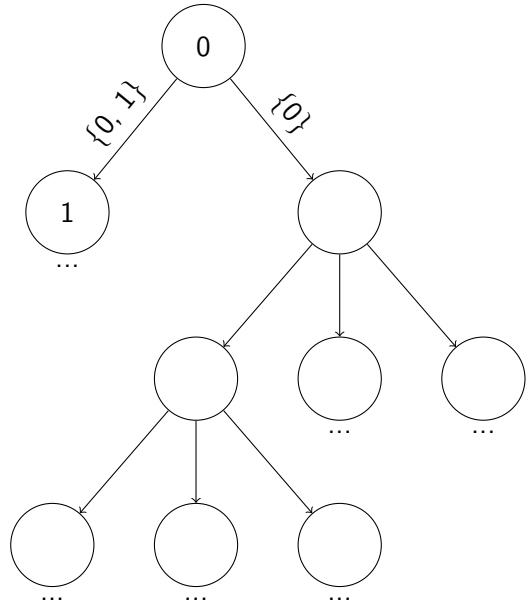
- Jednakże, P jest poprawny:
- gdy wszystkie wywołania rekurencyjne zwróciły dokładne wartości (tj. żadne nie zwróciło -1); wtedy flaga exact pozostanie ustawiona na true;
- w zakresie wartości nieprzekraczających $\max(R)$.
- Tylko gdy powyższe warunki są spełnione (w linii 14), result zostanie zapisany do TT (w linii 15) i zwrócony.
- W przeciwnym razie zostanie zwrócone -1 w linii 18. Wtedy s nie zawiera się w R, bo jest większe od $\max(R)$.

```
1  fun cut(s, R):
2      if s ∈ TT: return TT[s]
3      P ← {0, 1, ..., |N(s)|}
4      exact ← true
5      for m ∈ N(s):
6          if P ∩ R = ∅: return −1
7          v ← cut(m, (P \ {max(P)}) ∩
                  {0, ..., max(R)})
8          if v ∈ P:
9              P ← P \ {v}
10         else:
11             P ← P \ {max(P)}
12             if v = −1: exact ← false
13     result ← jedyny element w P
14     if exact or result ⩽ max(R):
15         TT[s] ← result
16         return result
17     else:
18         return −1
```

- Jednakże, P jest poprawny:
- gdy wszystkie wywołania rekurencyjne zwróciły dokładne wartości (tj. żadne nie zwróciło -1); wtedy flaga exact pozostanie ustawiona na true;
- w zakresie wartości nieprzekraczających $\max(R)$.
- Tylko gdy powyższe warunki są spełnione (w linii 14), result zostanie zapisany do TT (w linii 15) i zwrócony.
- W przeciwnym razie zostanie zwrócone -1 w linii 18. Wtedy s nie zawiera się w R, bo jest większe od $\max(R)$.

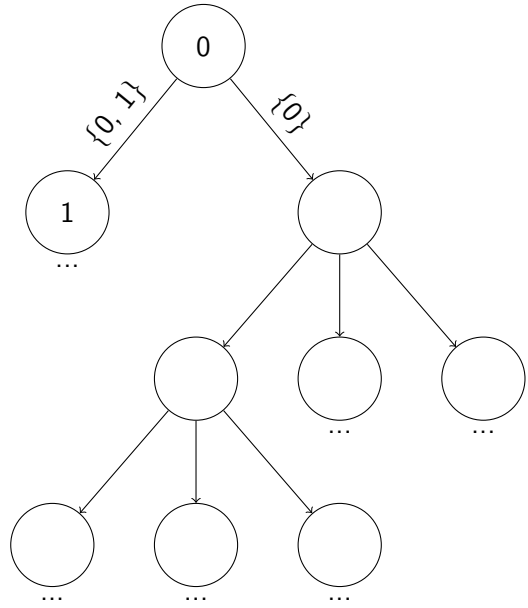
Odcięcia – przykładowy fragment drzewa poszukiwań

- Dodatkowe odcięcia zilustrujemy na tym samym drzewie co poprzednio.
- Przekazywany w dół drzewa wywołań zbiór (pokazany nad strzałkami) jest przecinany z $\{0, 1, \ldots, \max(R)\}$, gdzie R jest zbiorem otrzymanym od ojca.
- To zawęża do $\{0\}$ zbiór przekazany do następników prawego syna korzenia.
- W konsekwencji, algorytm może zakończyć badanie każdego z jego następników natychmiast po wyeliminowaniu 0 ze zbioru P (potencjalnych nimberów) tego następnika
- i wciąż jest w stanie dowieść, że nimber korzenia wynosi 0.



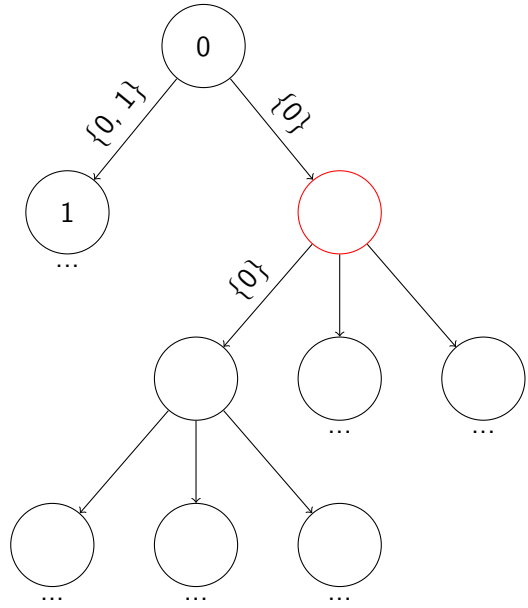
Odcięcia – przykładowy fragment drzewa poszukiwań

- Dodatkowe odcięcia zilustrujemy na tym samym drzewie co poprzednio.
- Przekazywany w dół drzewa wywołań zbiór (pokazany nad strzałkami) jest przecinany z $\{0, 1, \ldots, \max(R)\}$, gdzie R jest zbiorem otrzymanym od ojca.
- To zawęża do $\{0\}$ zbiór przekazany do następników prawego syna korzenia.
- W konsekwencji, algorytm może zakończyć badanie każdego z jego następników natychmiast po wyeliminowaniu 0 ze zbioru P (potencjalnych nimberów) tego następnika
- i wciąż jest w stanie dowieść, że nimber korzenia wynosi 0.



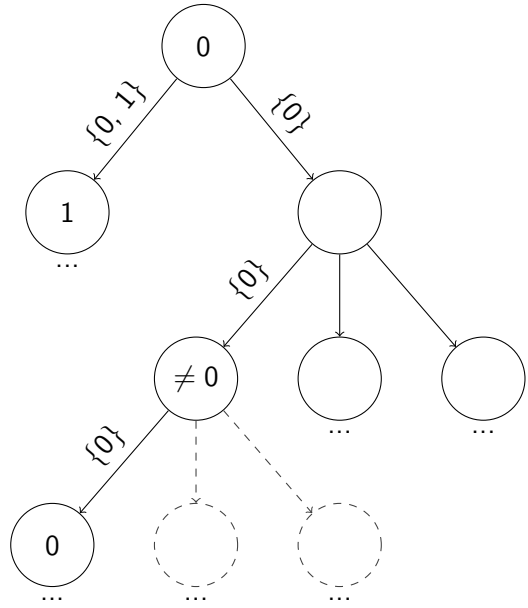
Odcięcia – przykładowy fragment drzewa poszukiwań

- Dodatkowe odcięcia zilustrujemy na tym samym drzewie co poprzednio.
- Przekazywany w dół drzewa wywołań zbiór (pokazany nad strzałkami) jest przecinany z $\{0, 1, \ldots, \max(R)\}$, gdzie R jest zbiorem otrzymanym od ojca.
- To zawęża do $\{0\}$ zbiór przekazany do następników prawego syna korzenia.
- W konsekwencji, algorytm może zakończyć badanie każdego z jego następników natychmiast po wyeliminowaniu 0 ze zbioru P (potencjalnych nimberów) tego następnika
- i wciąż jest w stanie dowieść, że nimber korzenia wynosi 0.



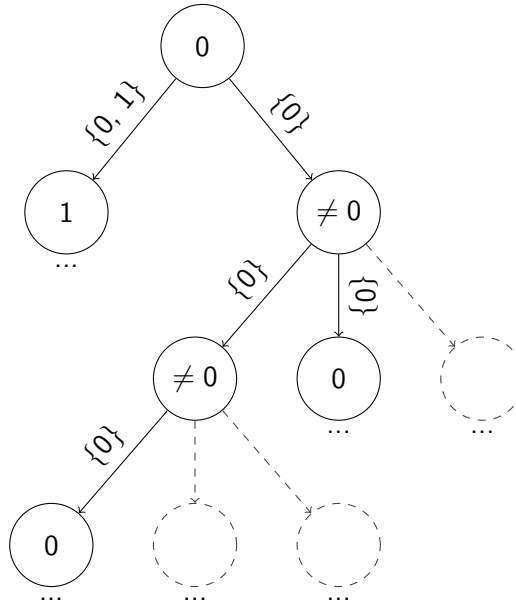
Odcięcia – przykładowy fragment drzewa poszukiwań

- Dodatkowe odcięcia zilustrujemy na tym samym drzewie co poprzednio.
- Przekazywany w dół drzewa wywołań zbiór (pokazany nad strzałkami) jest przecinany z $\{0, 1, \ldots, \max(R)\}$, gdzie R jest zbiorem otrzymanym od ojca.
- To zawęża do $\{0\}$ zbiór przekazany do następników prawego syna korzenia.
- W konsekwencji, algorytm może zakończyć badanie każdego z jego następników natychmiast po wyeliminowaniu 0 ze zbioru P (potencjalnych nimberów) tego następnika
- i wciąż jest w stanie dowieść, że nimber korzenia wynosi 0.



Odcięcia – przykładowy fragment drzewa poszukiwań

- Dodatkowe odcięcia zilustrujemy na tym samym drzewie co poprzednio.
- Przekazywany w dół drzewa wywołań zbiór (pokazany nad strzałkami) jest przecinany z $\{0, 1, \ldots, \max(R)\}$, gdzie R jest zbiorem otrzymanym od ojca.
- To zawęża do $\{0\}$ zbiór przekazany do następników prawego syna korzenia.
- W konsekwencji, algorytm może zakończyć badanie każdego z jego następników natychmiast po wyeliminowaniu 0 ze zbioru P (potencjalnych nimberów) tego następnika
- i wciąż jest w stanie dowieść, że nimber korzenia wynosi 0.



Enhanced Transposition Cut-off

- Enhanced Transposition Cut-off (ETC) to znane ulepszenie algorytmu α-β z tablicą transpozycji.
- Może ono być zaadoptowane do naszych metod w następujący sposób:
- Przed rekurencyjnym odwiedzeniem któregokolwiek z następników w głównej pętli,
- ETC wyszukuje ich nimberów w TT,
- by wykluczyć możliwie dużo wartości z P,
- co nawet może zakończyć się odcięciem.
- Następnie, w głównej pętli przeglądane są tylko te następniki, których ETC nie znalazło w TT.

Enhanced Transposition Cut-off

- Enhanced Transposition Cut-off (ETC) to znane ulepszenie algorytmu α-β z tablicą transpozycji.
- Może ono być zaadoptowane do naszych metod w następujący sposób:
- Przed rekurencyjnym odwiedzeniem któregokolwiek z następników w głównej pętli,
- ETC wyszukuje ich nimberów w TT,
- by wykluczyć możliwie dużo wartości z P,
- co nawet może zakończyć się odcięciem.
- Następnie, w głównej pętli przeglądane są tylko te następniki, których ETC nie znalazło w TT.

Enhanced Transposition Cut-off

- Enhanced Transposition Cut-off (ETC) to znane ulepszenie algorytmu α-β z tablicą transpozycji.
- Może ono być zaadoptowane do naszych metod w następujący sposób:
- Przed rekurencyjnym odwiedzeniem któregokolwiek z następników w głównej pętli,
- ETC wyszukuje ich nimberów w TT,
- by wykluczyć możliwie dużo wartości z P,
- co nawet może zakończyć się odcięciem.
- Następnie, w głównej pętli przeglądane są tylko te następniki, których ETC nie znalazło w TT.

Enhanced Transposition Cut-off

- Enhanced Transposition Cut-off (ETC) to znane ulepszenie algorytmu α-β z tablicą transpozycji.
- Może ono być zaadoptowane do naszych metod w następujący sposób:
- Przed rekurencyjnym odwiedzeniem któregokolwiek z następników w głównej pętli,
- ETC wyszukuje ich nimberów w TT,
- by wykluczyć możliwie dużo wartości z P,
- co nawet może zakończyć się odcięciem.
- Następnie, w głównej pętli przeglądane są tylko te następniki, których ETC nie znalazło w TT.

Enhanced Transposition Cut-off

- Enhanced Transposition Cut-off (ETC) to znane ulepszenie algorytmu α-β z tablicą transpozycji.
- Może ono być zaadoptowane do naszych metod w następujący sposób:
- Przed rekurencyjnym odwiedzeniem któregokolwiek z następników w głównej pętli,
- ETC wyszukuje ich nimberów w TT,
- by wykluczyć możliwie dużo wartości z P,
- co nawet może zakończyć się odcięciem.
- Następnie, w głównej pętli przeglądane są tylko te następniki, których ETC nie znalazło w TT.

Enhanced Transposition Cut-off

- Enhanced Transposition Cut-off (ETC) to znane ulepszenie algorytmu α-β z tablicą transpozycji.
- Może ono być zaadoptowane do naszych metod w następujący sposób:
- Przed rekurencyjnym odwiedzeniem któregokolwiek z następników w głównej pętli,
- ETC wyszukuje ich nimberów w TT,
- by wykluczyć możliwie dużo wartości z P,
- co nawet może zakończyć się odcięciem.
- Następnie, w głównej pętli przeglądane są tylko te następniki, których ETC nie znalazło w TT.

Enhanced Transposition Cut-off

- Enhanced Transposition Cut-off (ETC) to znane ulepszenie algorytmu α-β z tablicą transpozycji.
- Może ono być zaadoptowane do naszych metod w następujący sposób:
- Przed rekurencyjnym odwiedzeniem któregokolwiek z następników w głównej pętli,
- ETC wyszukuje ich nimberów w TT,
- by wykluczyć możliwie dużo wartości z P,
- co nawet może zakończyć się odcięciem.
- Następnie, w głównej pętli przeglądane są tylko te następniki, których ETC nie znalazło w TT.

Aspirujące zbiory

- Przedstawione algorytmy cięć zwracają nimbera, gdy jest on w zbiorze przekazanym jako drugi argument.
- Jeśli nie jest, to mogą równie dobrze zwrócić -1.
- Dlatego przekazanie $\{0, 1, ..., |N(s)|\}$ gwarantuje zwrócenie nimbera:

```
1  fun nimber_of(s):
2    return cut(s, {0, 1, ..., N(s)})
```

- Metoda *aspirujących zbiorów* polega na przekazywaniu mniejszych zbiorów, aż do uzyskania nimbera:

```
1  fun aspset_nimber_of(s):
2      for potencjalny_nimber ← 0, 1, ...:
3          v ← cut(s, {potencjalny_nimber})
4          if v ≠ −1: return v
```

Pomimo potencjalnie większej liczby wywołań przeszukiwania, taka metoda może się opłacać, ponieważ użycie mniejszych zbiorów zazwyczaj prowadzi do znacznie większej liczby odcięć.

Aspirujące zbiory

- Przedstawione algorytmy cięć zwracają nimbera, gdy jest on w zbiorze przekazanym jako drugi argument.
- Jeśli nie jest, to mogą równie dobrze zwrócić -1.
- Dlatego przekazanie $\{0, 1, ..., |N(s)|\}$ gwarantuje zwrócenie nimbera:

```
1  fun nimber_of(s):
2    return cut(s, {0, 1, ..., N(s)})
```

- Metoda *aspirujących zbiorów* polega na przekazywaniu mniejszych zbiorów, aż do uzyskania nimbera:

```
1  fun aspset_nimber_of(s):
2      for potencjalny_nimber ← 0, 1, ...:
3          v ← cut(s, {potencjalny_nimber})
4          if v ≠ −1: return v
```

Pomimo potencjalnie większej liczby wywołań przeszukiwania, taka metoda może się opłacać, ponieważ użycie mniejszych zbiorów zazwyczaj prowadzi do znacznie większej liczby odcięć.

Aspirujące zbiory

- Przedstawione algorytmy cięć zwracają nimbera, gdy jest on w zbiorze przekazanym jako drugi argument.
- Jeśli nie jest, to mogą równie dobrze zwrócić -1.
- Dlatego przekazanie $\{0, 1, ..., |N(s)|\}$ gwarantuje zwrócenie nimbera:

```
1  fun nimber_of(s):
2    return cut(s, {0, 1, ..., N(s)})
```

- Metoda *aspirujących zbiorów* polega na przekazywaniu mniejszych zbiorów, aż do uzyskania nimbera:

```
1  fun aspset_nimber_of(s):
2      for potencjalny_nimber ← 0, 1, ...:
3          v ← cut(s, {potencjalny_nimber})
4          if v ≠ −1: return v
```

Pomimo potencjalnie większej liczby wywołań przeszukiwania, taka metoda może się opłacać, ponieważ użycie mniejszych zbiorów zazwyczaj prowadzi do znacznie większej liczby odcięć.

Aspirujące zbiory

- Przedstawione algorytmy cięć zwracają nimbera, gdy jest on w zbiorze przekazanym jako drugi argument.
- Jeśli nie jest, to mogą równie dobrze zwrócić -1.
- Dlatego przekazanie $\{0, 1, ..., |N(s)|\}$ gwarantuje zwrócenie nimbera:

```
1  fun nimber_of(s):
2    return cut(s, {0, 1, ..., N(s)})
```

- Metoda *aspirujących zbiorów* polega na przekazywaniu mniejszych zbiorów, aż do uzyskania nimbera:

```
1  fun aspset_nimber_of(s):
2      for potencjalny_nimber ← 0, 1, ...:
3          v ← cut(s, {potencjalny_nimber})
4          if v ≠ −1: return v
```

Pomimo potencjalnie większej liczby wywołań przeszukiwania, taka metoda może się opłacać, ponieważ użycie mniejszych zbiorów zazwyczaj prowadzi do znacznie większej liczby odcięć.

Bibliografia

- P. Beling, M. Rogalski, *On pruning search trees of impartial games*, Artificial Intelligence 283, 2020
- R. P. Sprague, *Über mathematische Kampfspiele*, 1935-36, Tohoku Mathematical Journal, 41: 438-444
- P. M. Grundy *Mathematics and games*, 1939, Eureka, 2: 6-8
- E. R. Berlekamp, J. H. Conway, R. K. Guy, *Winning Ways for your Mathematical Plays*, 2001-2004, Vol. I-IV
- Donald E. Knuth, Ronald W. Moore, *An Analysis of Alpha-Beta Pruning*, Artificial Intelligence 6(4), 1975
- A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin, *Nearly optimal minimax tree search?*, Technical report, Department of Computing Science, The University of Alberta, Edmonton, Alberta, Canada, 1994
- A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin, *Exploiting graph properties of game trees*, in: W.J. Clancey, D.S. Weld (Eds.), Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, vol. 1, AAAI 96, IAAI 96, AAAI Press/The MIT Press, Portland, Oregon, 1996, pp. 234–239