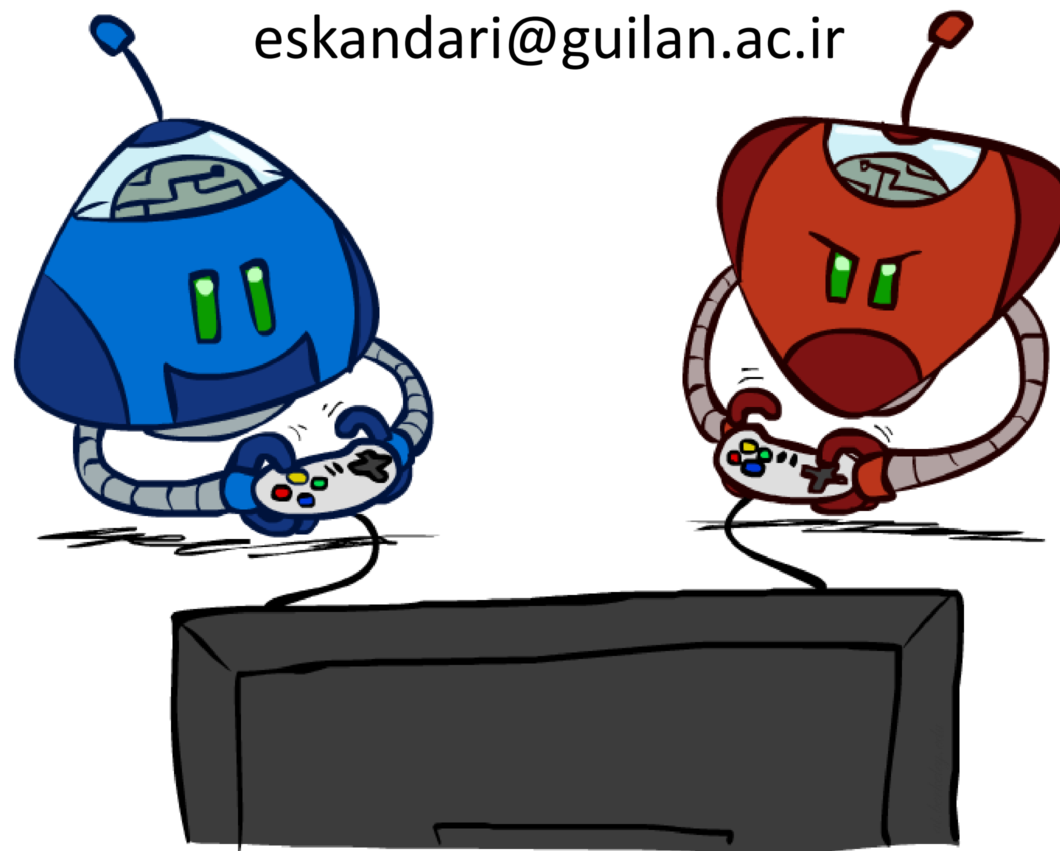
# هوش مصنوعی
# (جستجو در حضور عامل های دیگر-بخش اول)

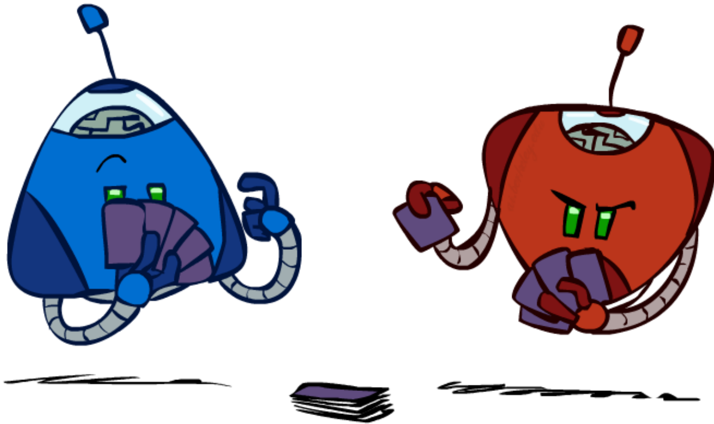
صادق اسکندری - دانشکده علوم ریاضی، گروه علوم کامپیوتر

eskandari@guilan.ac.ir

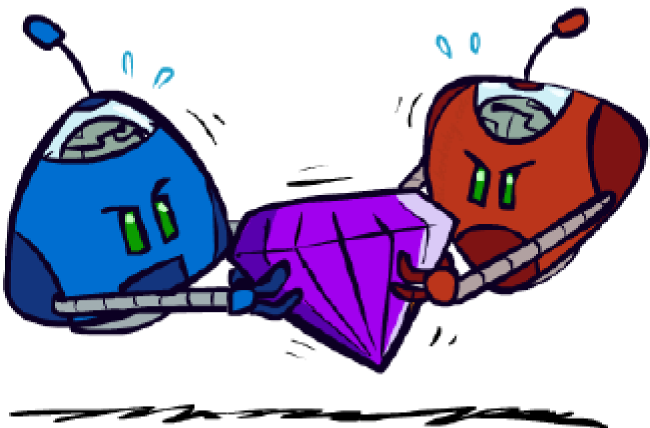


# مقدمه

- تابحال مفاهیم جستجو در محیط های تک عاملی (Single Agent) مورد بررسی قرار گرفتند.
- محیط هایی که بیش از یک عامل در آن عمل کرده و بر یکدیگر تأثیر می گذارند را محیط های چند عاملی (Multi-Agent) می گویند.
- انواع محیط های چند عاملی
  - محیط های همکاری: اهداف عامل ها با یکدیگر همسو است.
  - محیط های رقابتی: اهداف عامل ها با یکدیگر در تناقض است.



- به جستجو در محیط های چند عاملی رقابتی، بازی (Game) گفته می شود.
- بازی ها انواع متنوعی دارند. ولی نوعی که در این بخش به آن خواهیم پرداخت عبارت است از:
  - محیط های قطعی و دسترس پذیر
  - تعداد دو عامل به صورت نوبتی عمل می کنند.
  - میزان تأثیر برد و باخت برای هر دو عامل یکسان است (Zero-Sum Game)



# مقدمه

## موقعیت فعلی برخی از بازیهای معروف

### چکرز (Checkers)

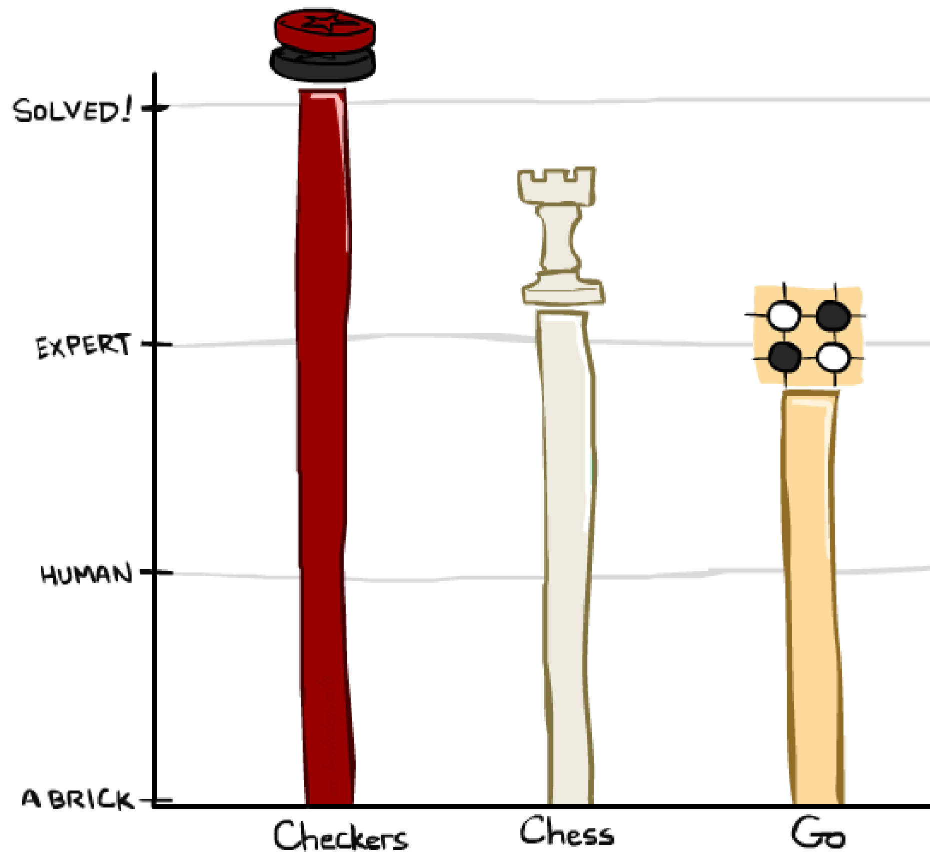
در سال ۱۹۹۶ کامپیوتر توانست قهرمان جهان (Tinsley) را شکست دهد.
در سال ۲۰۰۷ این بازی کاملاً حل شده است.

### شطرنج (Chess)

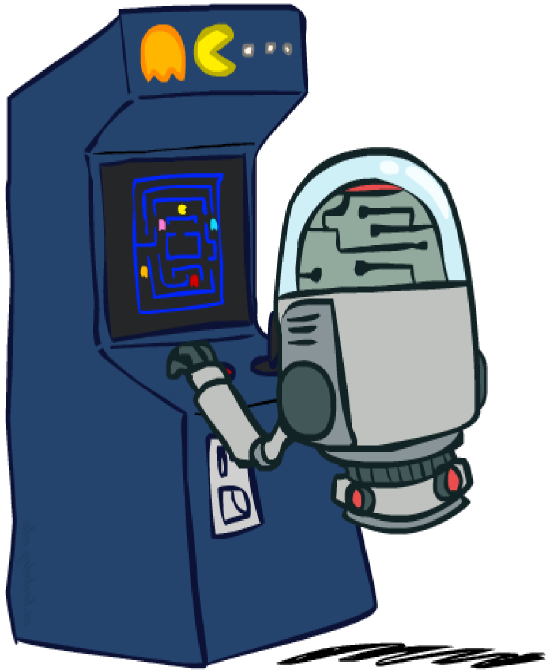
در سال ۱۹۹۷، کامپیوتر (DeepBlue) توانست قهرمان جهان ( Gary Kasparov) را شکست دهد. DeepBlue قادر بود تا 200M حالت را در ثانیه بررسی کند.

### گو (Go)

در سال ۲۰۱۶، کامپیوتر (AlphaGo) توانست قهرمان جهان (از کره جنوبی) را شکست دهد. در ساختار AlphaGo از درخت جستجوی MontoCarlo و یادگیری ماشینی استفاده شده است.



# فرموله سازی جستجو در بازی



برای تعریف یک بازی، به اطلاعات زیر نیاز داریم:

۱- حالت اولیه ۲- اعمال ممکن (A) ۳- بازیکنان (P)

٤- تابع بعدی $\text{Successor}: S \times A \rightarrow S$

٥- تابع تست خاتمه $\text{Terminal-Test}: S \rightarrow \{T, F\}$

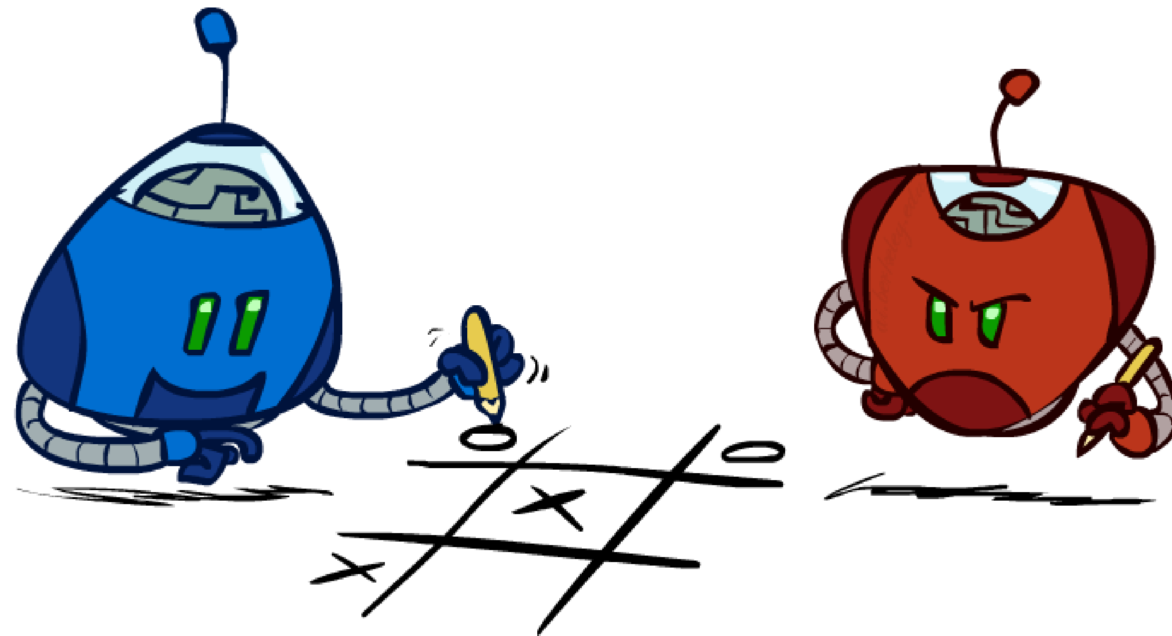
٦- تابع سودمندی حالت پایانی (Terminal Utility)؛ مقداری را برای هر حالت پایانی و برای هر بازیکن در نظر می گیرد. $\text{Terminal-Utility}: S \times P \rightarrow R$

حالت اولیه و حرکت های ممکن برای هر بازیکن، درخت بازی (Game Tree) را مشخص می کند.

# فرموله سازی جستجو در بازی

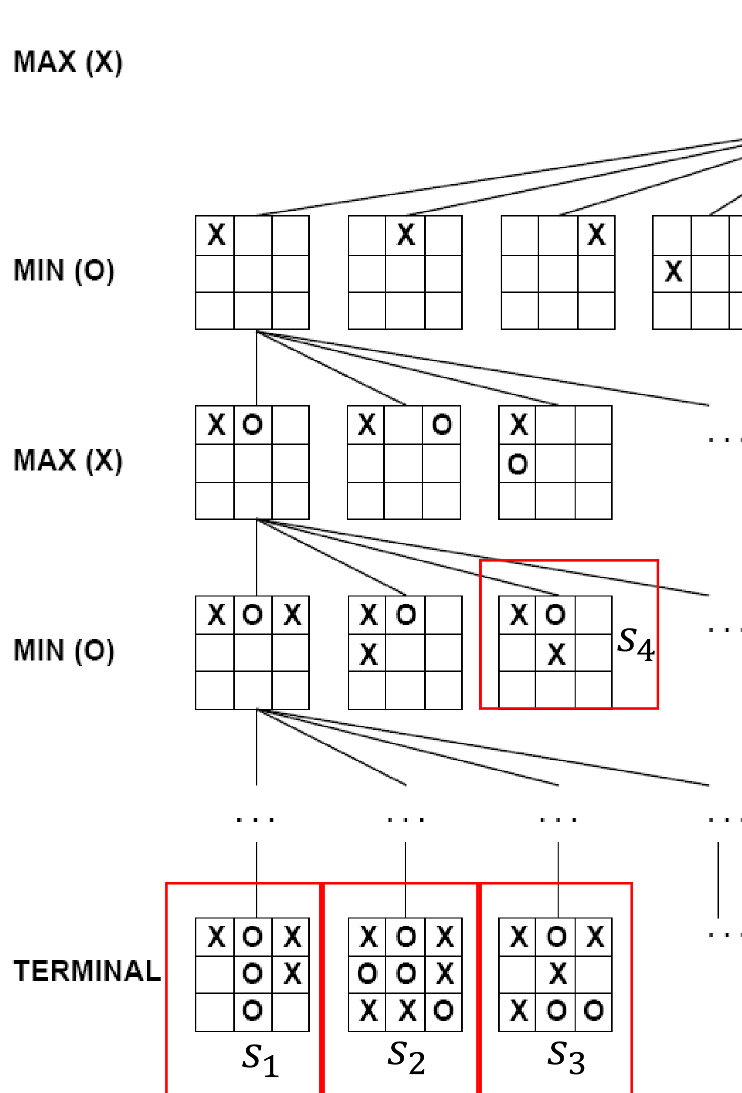
## مثال: بازی Tic-Tac-Toe

در این بازی، با یک جدول $3 \times 3$ مواجه هستیم. یکی از بازیکنان از علامت $\times$ و دیگری از علامت O استفاده می کند. اگر در نهایت یکی از بازیکنان بتواند علامت های خودرا به شکل افقی، عمودی یا قطری بچیند، برنده بازی خواهد بود.



# فرموله سازی جستجو در بازی

## مثال: بازی Tic-Tac-Toe




حالت اولیه: صفحه $3 \times 3$ خالی

بازیکنان: {MAX,MIN}

اعمال و تابع بعدی

تابع تست خاتمه: آیا برای هر حالت داده شده، یکی از بازیکنان توانسته مهره های خود را به شکل افقی، عمودی و یا قطری بچینند؟ و یا اینکه آیا کل خانه های جدول پر شده اند؟

$$terminal-test(s_1) = True$$
$$terminal-test(s_4) = False$$

تابع سودمندی:

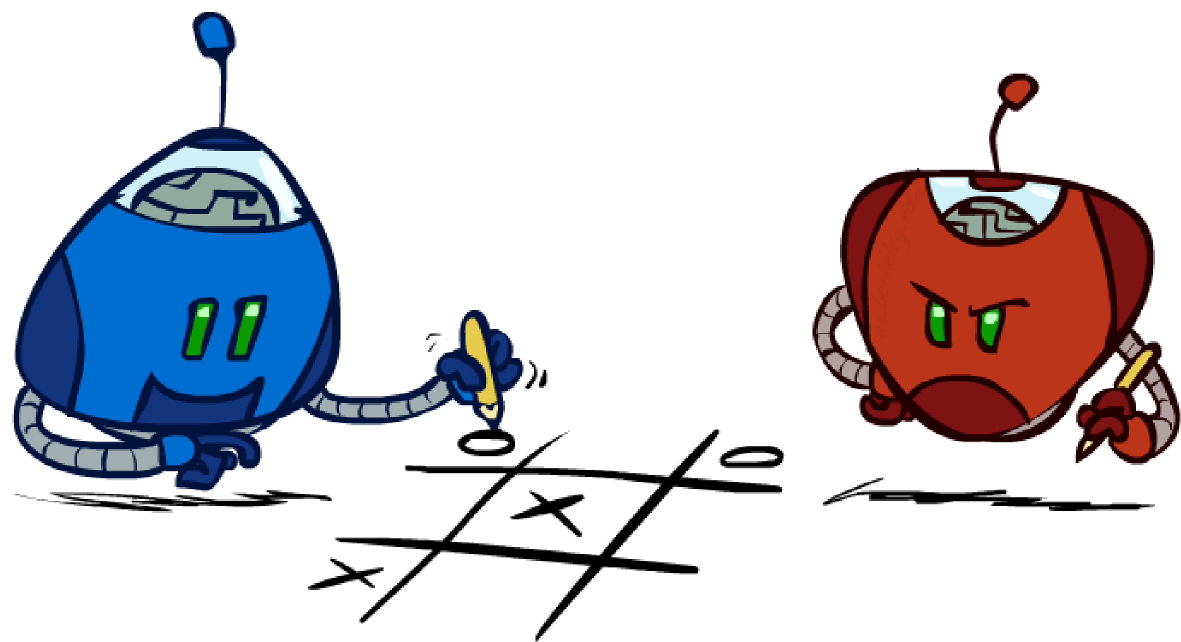
$$terminal-utility(s_1, MAX) = -1$$
$$terminal-utility(s_1, MIN) = +1$$
$$terminal-utility(s_2, MAX) = 0$$
$$terminal-utility(s_2, MIN) = 0$$

$$terminal-utility(s_3, MAX) = +1$$
$$terminal-utility(s_3, MIN) = -1$$

$$terminal-utility(s_4, MAX) = ?$$

# جستجو برای مسائل بازی

بر خلاف مسائل جستجوی برنامه ریزی، که عامل یک دنباله از اعمال را از ابتدا می یافت، در یک بازی MAX باید بر اساس حرکت های MIN عمل بعدی خود را انتخاب کند. بنابراین، از همان ابتدا نمی توان یک دنباله حرکت را معرفی کرد. در نتیجه باید به پرسش های زیر پاسخ داد:



۱- بهترین حرکت MAX در حالت اولیه چیست؟

۲- بهترین حرکت MAX بعد از حرکت اول MIN چیست؟

۳- بهترین حرکت MAX بعد از حرکت دوم MIN چیست؟

⋮

# جستجو برای مسائل بازی: الگوریتم MINIMAX

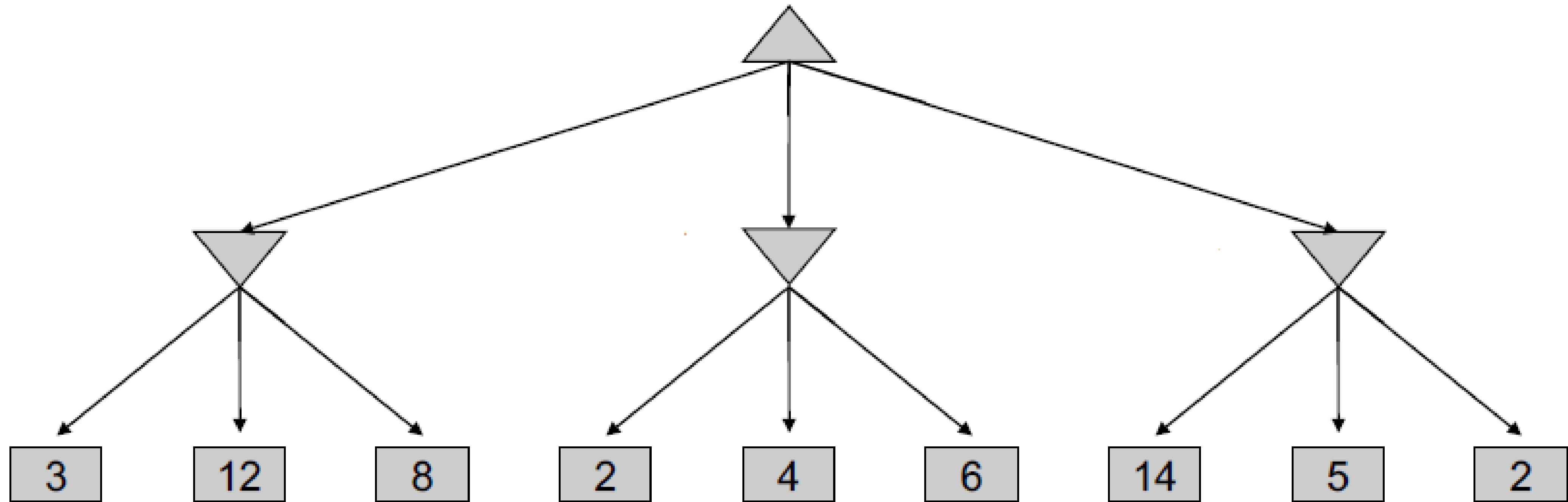
در الگوریتم MINIMAX به هر گره درخت بازی یک مقدار تحت عنوان minimax-value اختصاص داده می شود.

$$minimax-value(n)=\begin{cases} Terminal-Utility(n,MAX) & \text{if n is a terminal node} \\ max_{s\in successors(n)}\left(minimax-value(s)\right) & \text{if n is a MAX node} \\ min_{s\in successors(n)}\left(minimax-value(s)\right) & \text{if n is a MIN node} \end{cases}$$

تمامی مقادیر فوق از دیدگاه MAX محاسبه می شوند.

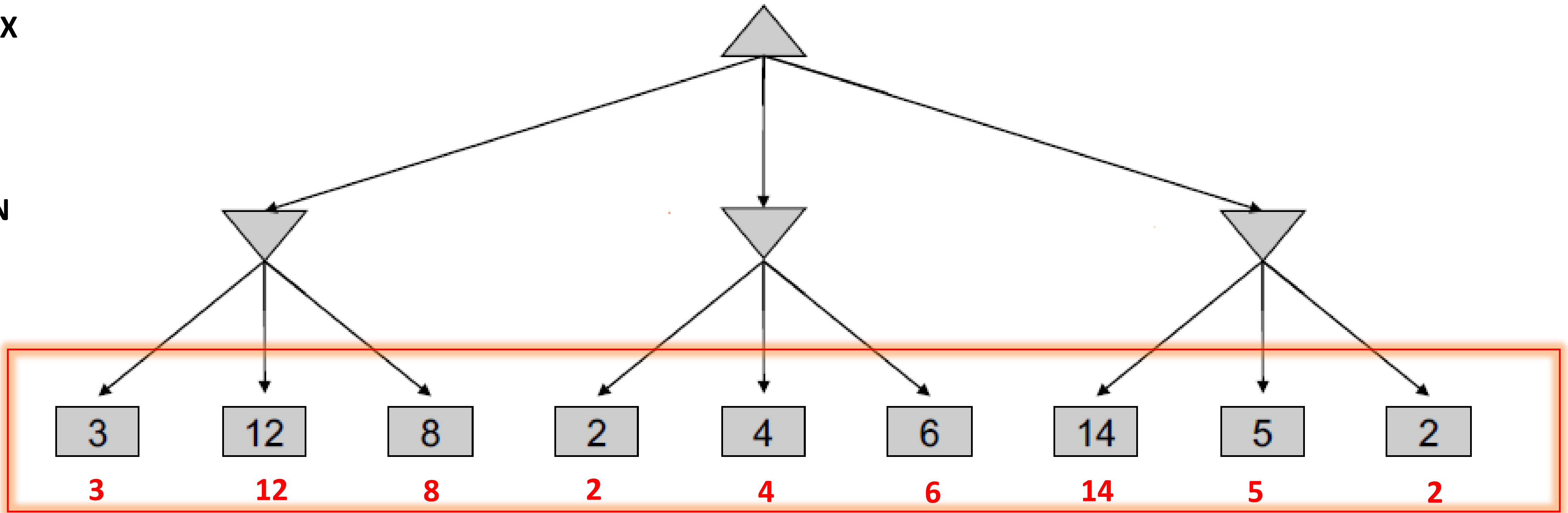
# جستجو برای مسائل بازی: الگوریتم MINIMAX

مثال: مقادیر $minimax$ را برای هر یک از گرههای درخت بازی زیر محاسبه کنید:



# جستجو برای مسائل بازی: الگوریتم MINIMAX

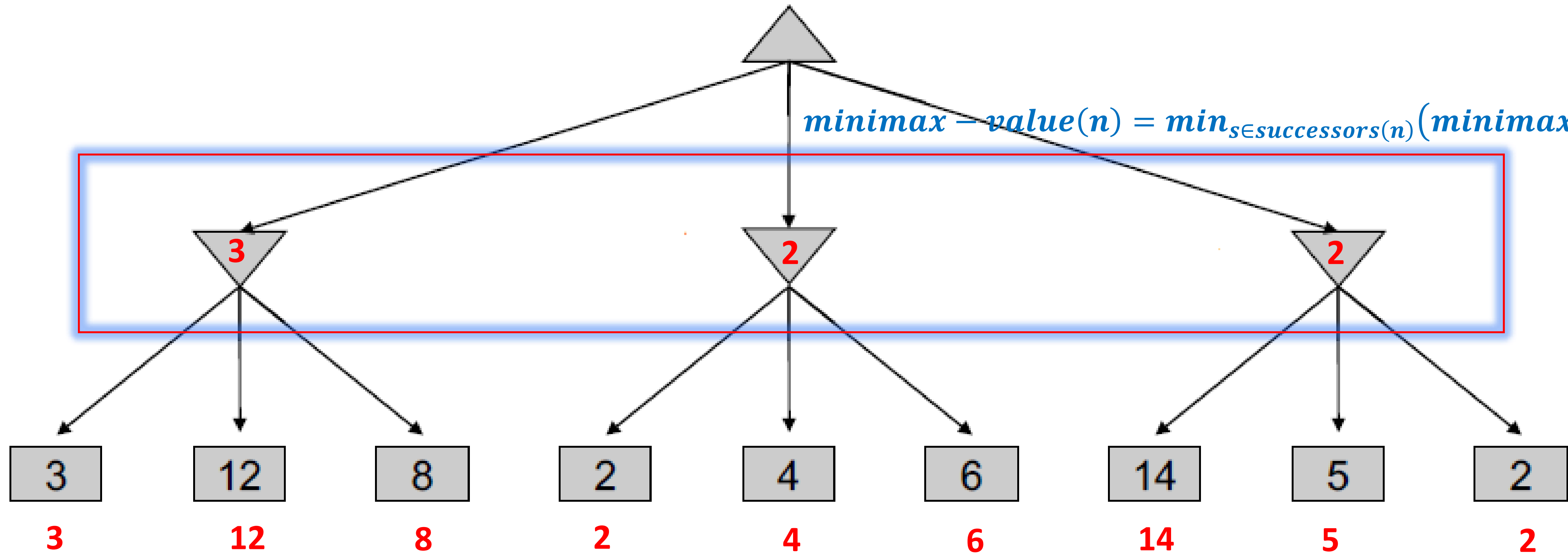
مثال: مقادیر $minimax$ را برای هر یک از گرههای درخت بازی زیر محاسبه کنید:




$$minimax-value(n) = Terminal-Utility(n, MAX)$$

# جستجو برای مسائل بازی: الگوریتم MINIMAX

مثال: مقادیر $minimax$ را برای هر یک از گرههای درخت بازی زیر محاسبه کنید:

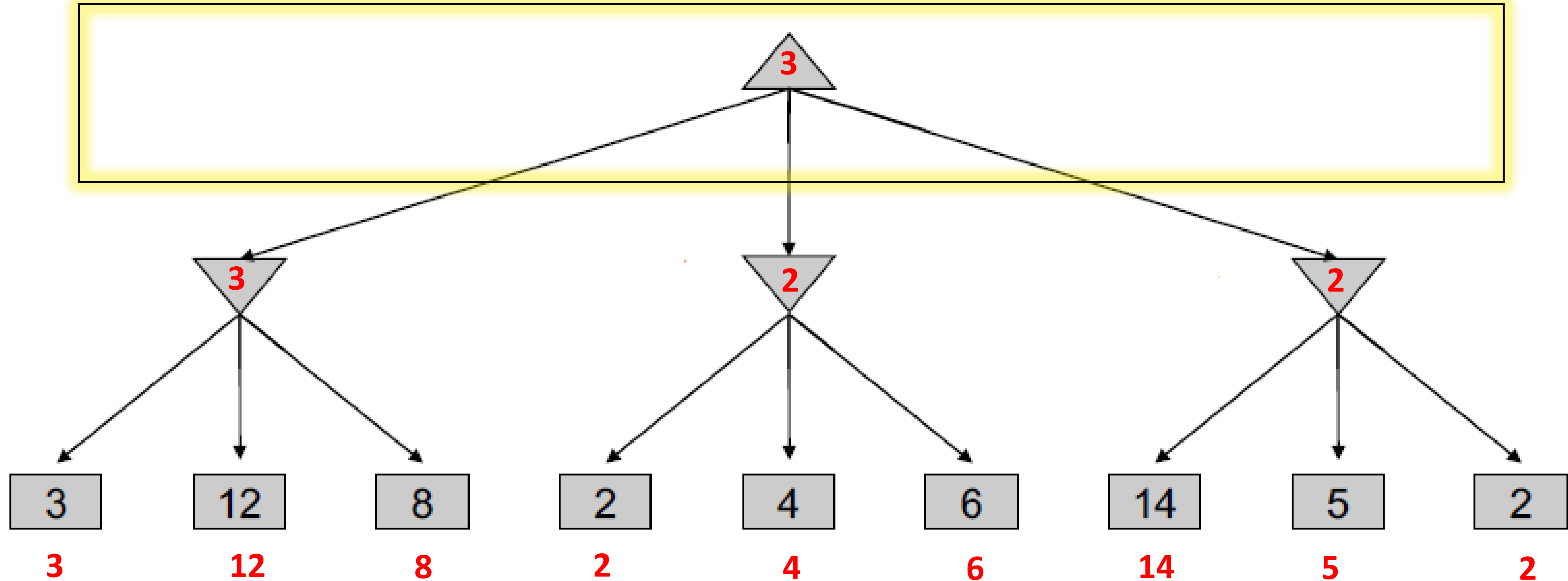



$$\boldsymbol{minimax-value(n) = min_{s \in successors(n)}(minimax-value(s))}$$

# جستجو برای مسائل بازی: الگوریتم MINIMAX

مثال: مقادیر $minimax$ را برای هر یک از گره‌های درخت بازی زیر محاسبه کنید:

$$minimax - value(n) = max_{s \in successors(n)}(minimax - value(s))$$



# جستجو برای مسائل بازی: الگوریتم MINIMAX

پیاده سازی الگوریتم محاسبه $minimax - value$

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

# جستجو برای مسائل بازی: الگوریتم MINIMAX

## کارآیی الگوریتم MiniMax

این الگوریتم مشابه الگوریتم DFS است.

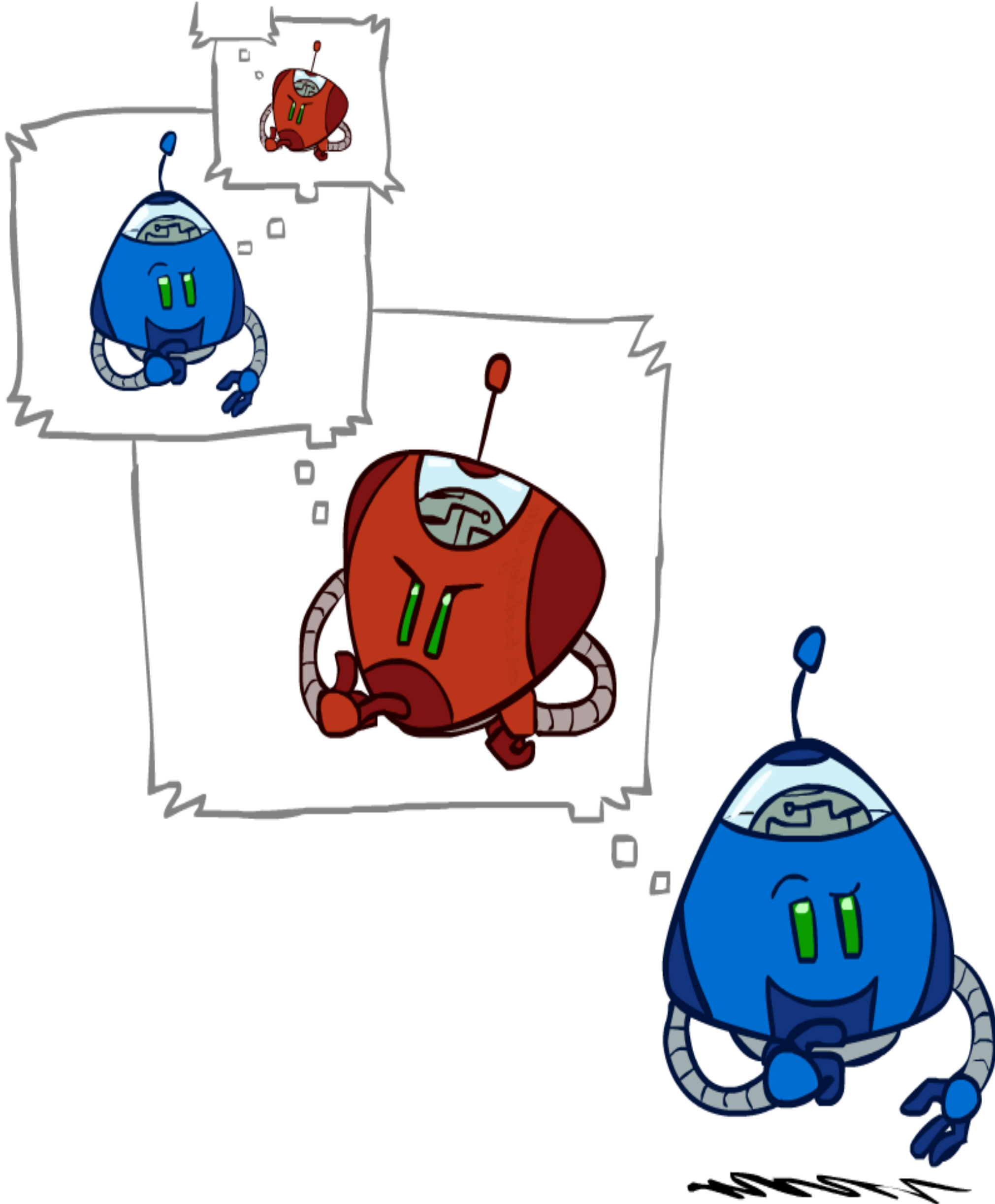
پیچیدگی زمانی: $O(b^m)$

$m$: حداکثر عمق

$b$: فاکتور انشعاب

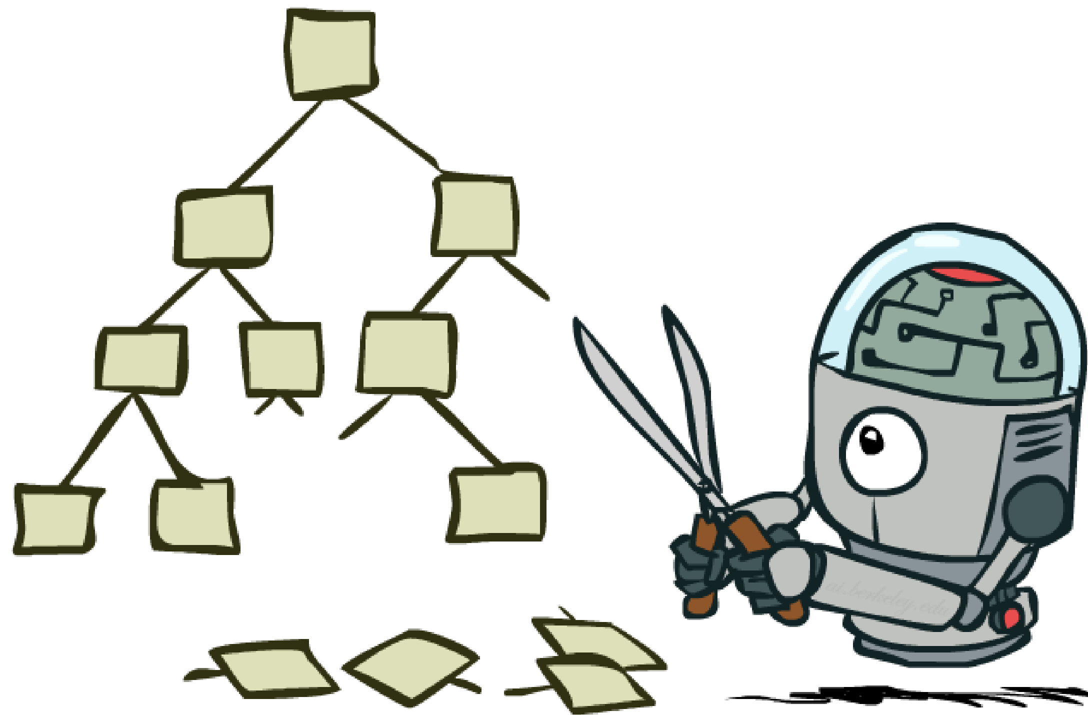
پیچیدگی فضایی: $O(bm)$

برای بازی شطرنج، $b \approx 35$ و $m \approx 100$ و در نتیجه الگوریتم minimax دارای پیچیدگی زمانی $O(35^{100})$ خواهد بود. ☹



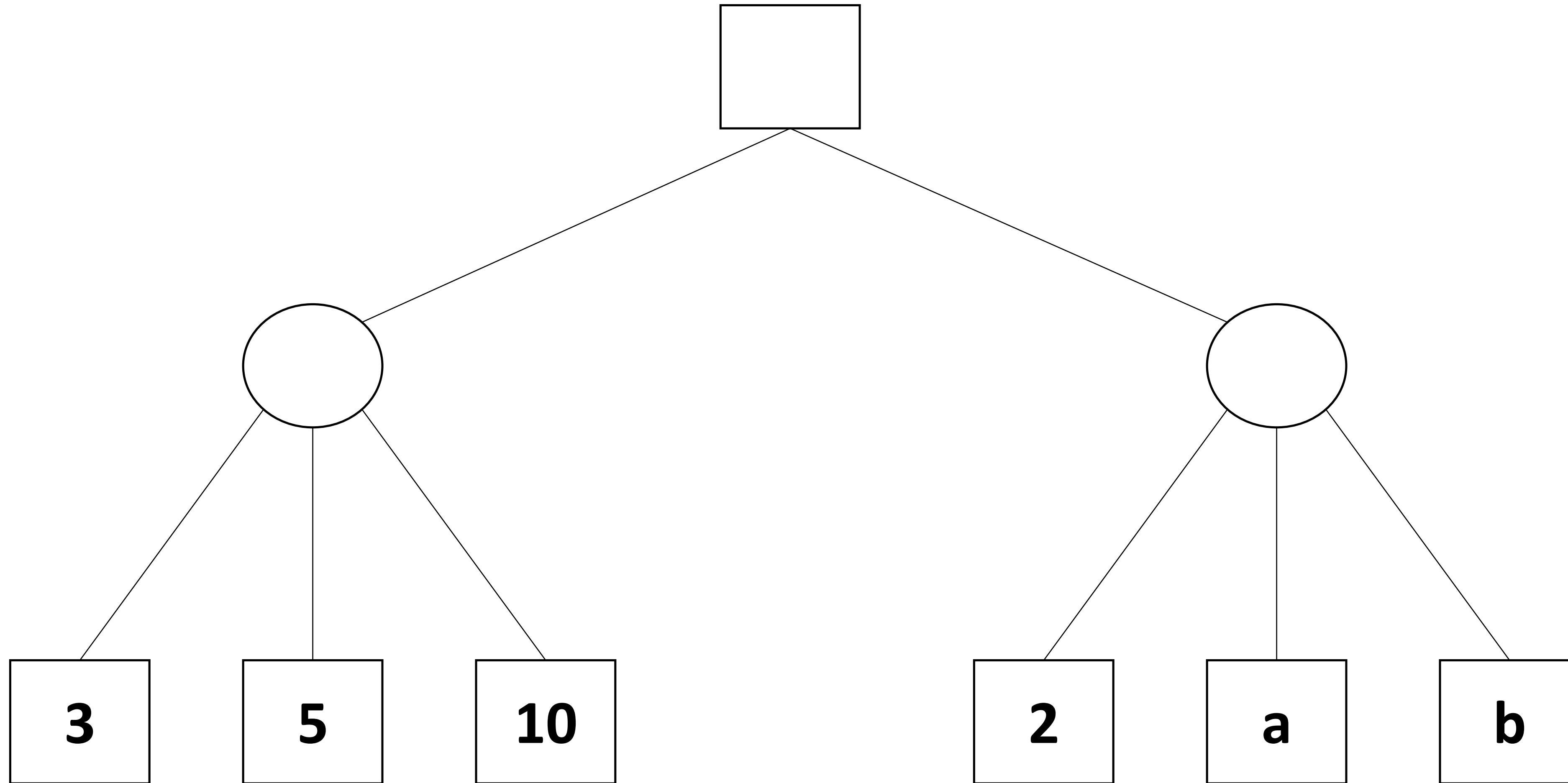
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

در هرس $\alpha - \beta$، می خواهیم از محاسبه $minimax - value$ برای زیردرخت هایی که مقدارشان تأثیری در نتیجه نهایی ندارد، جلوگیری کنیم.



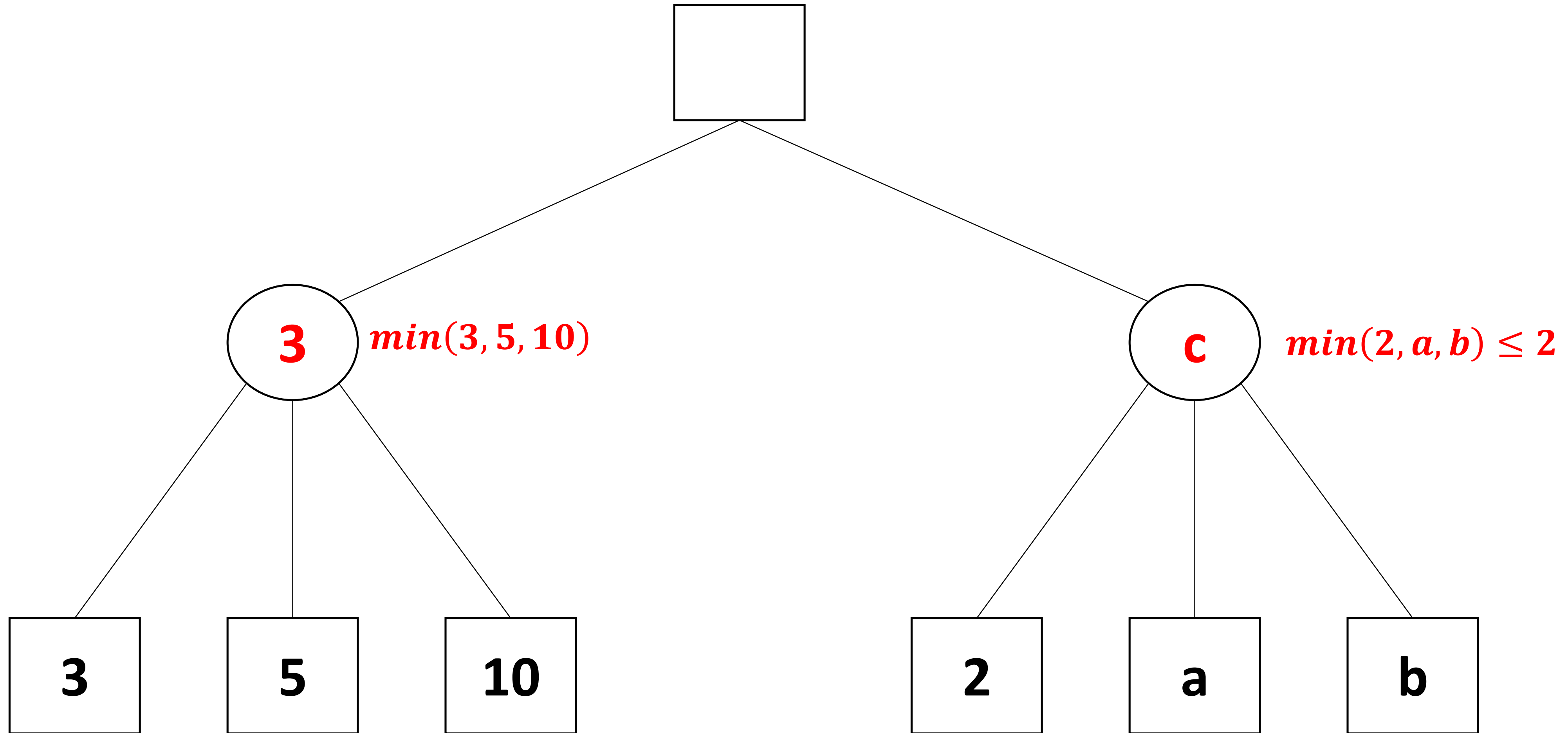
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را در نظر بگیرید:



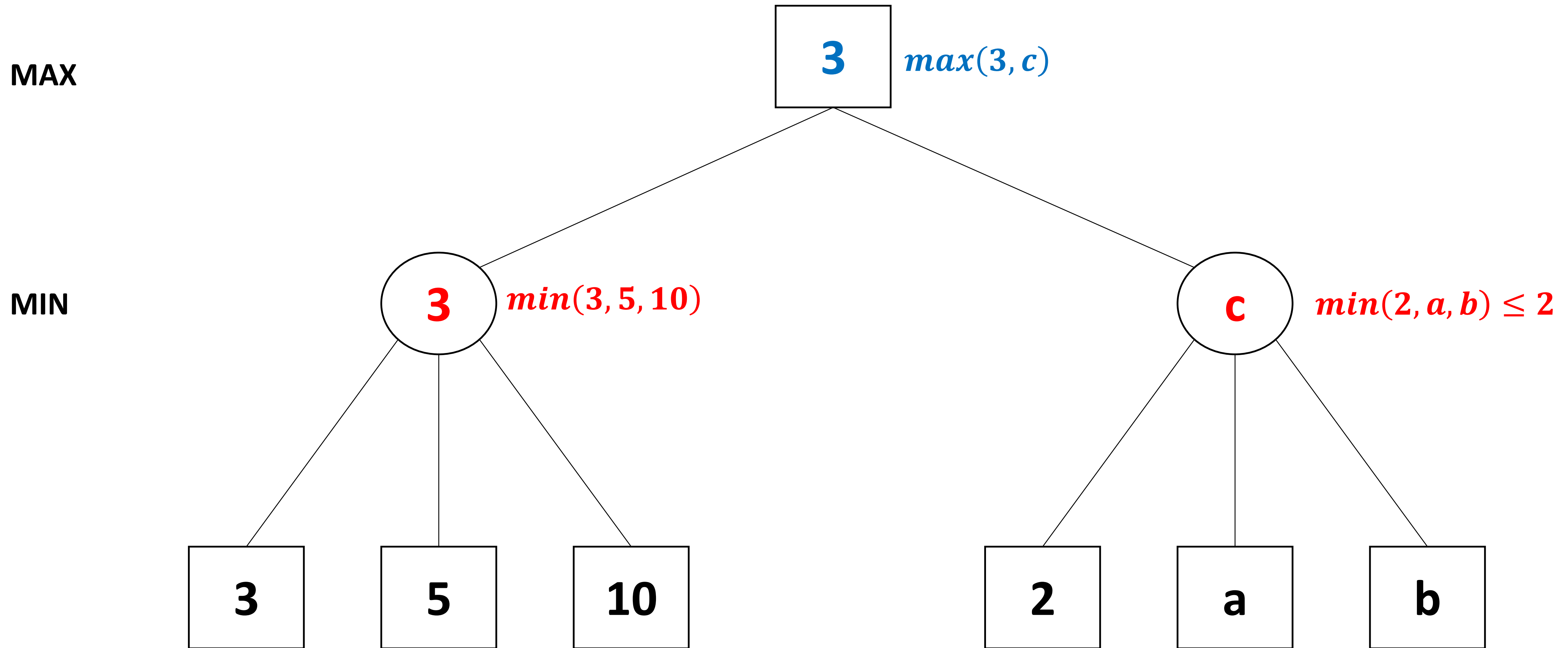
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را در نظر بگیرید:



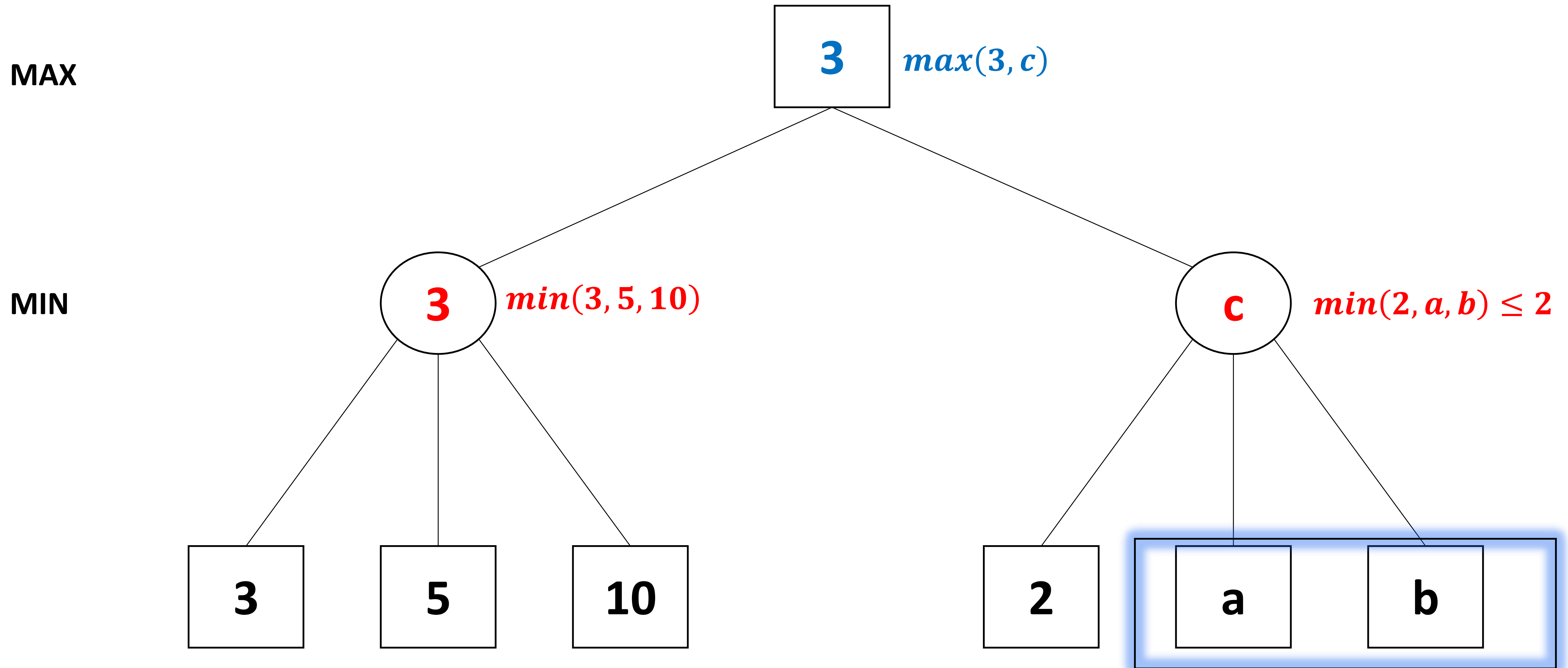
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را در نظر بگیرید:



# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را در نظر بگیرید:



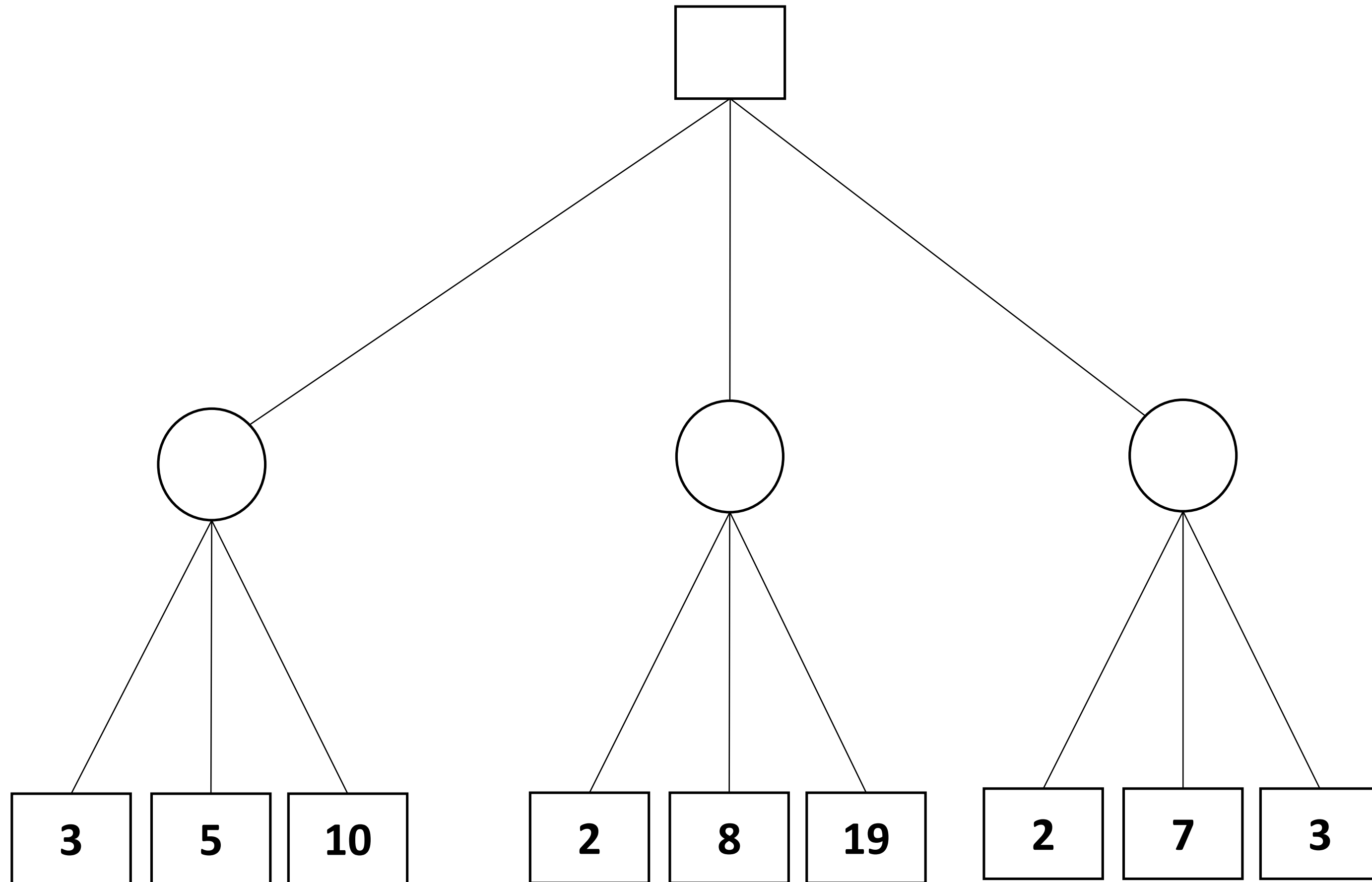

نیازی به محاسبه مقادیر a و b نیست ☺

# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

- آلفا ($\alpha$): بهترین انتخاب برای بازیکن MAX تابحال (می خواهیم این مقدار تا جایی که امکان دارد بزرگ باشد)
- بتا ($\beta$): بهترین انتخاب برای بازیکن MIN تابحال (می خواهیم این مقدار تا جایی که امکان دارد کوچک باشد)
- هر گره مقادیر آلفا و بتا خود را دارد.
- مقدار آلفا تنها توسط گرههای MAX و مقدار بتا توسط گرههای MIN عوض می شوند.
- یک گره زمانی هرس می شود که مقدار آلفا آن از مقدار بتا بیشتر باشد.
- مقدار اولیه برای گره ریشه به صورت $\boldsymbol{\alpha} = -\infty$ و $\boldsymbol{\beta} = +\infty$ می باشد.

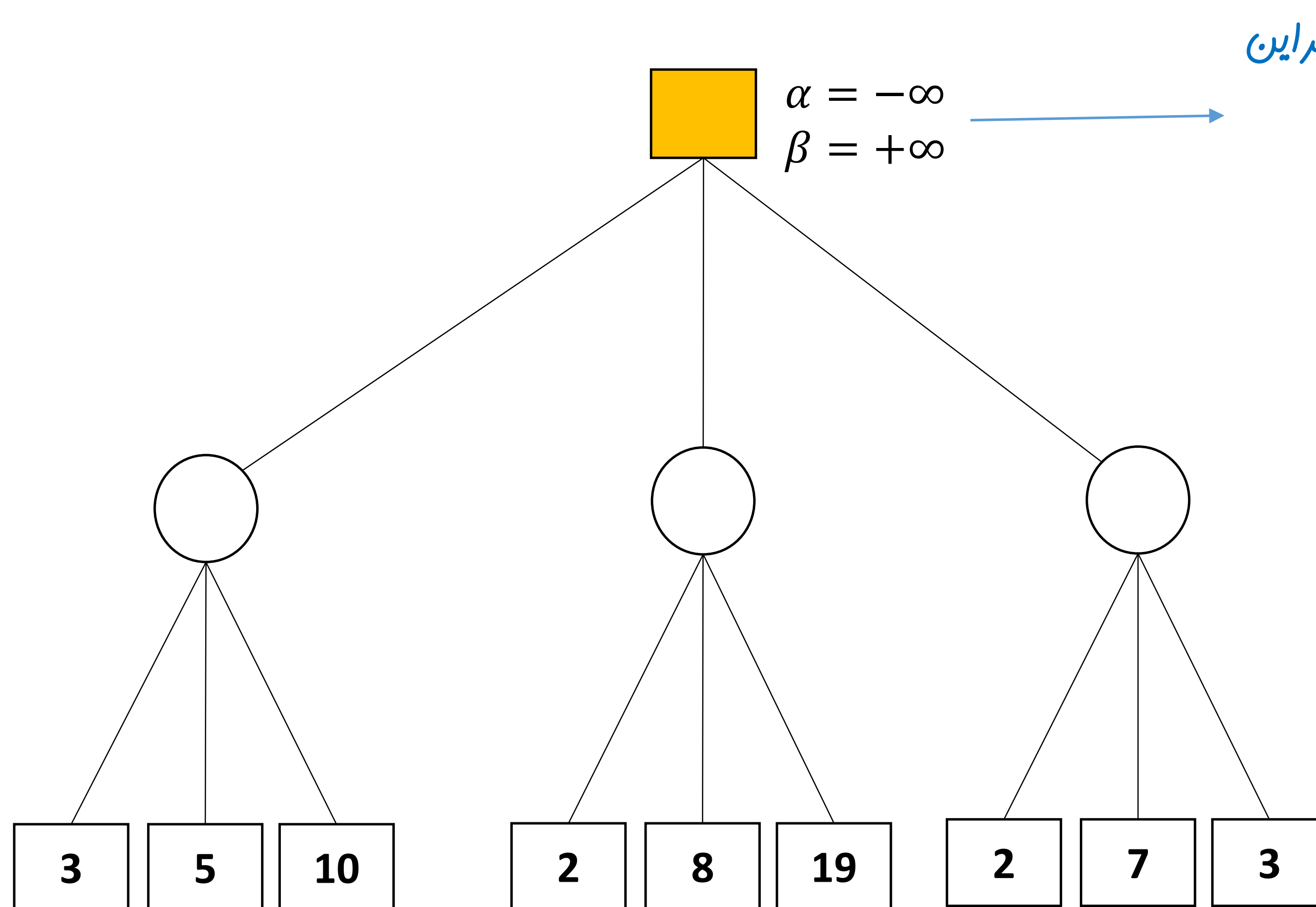
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

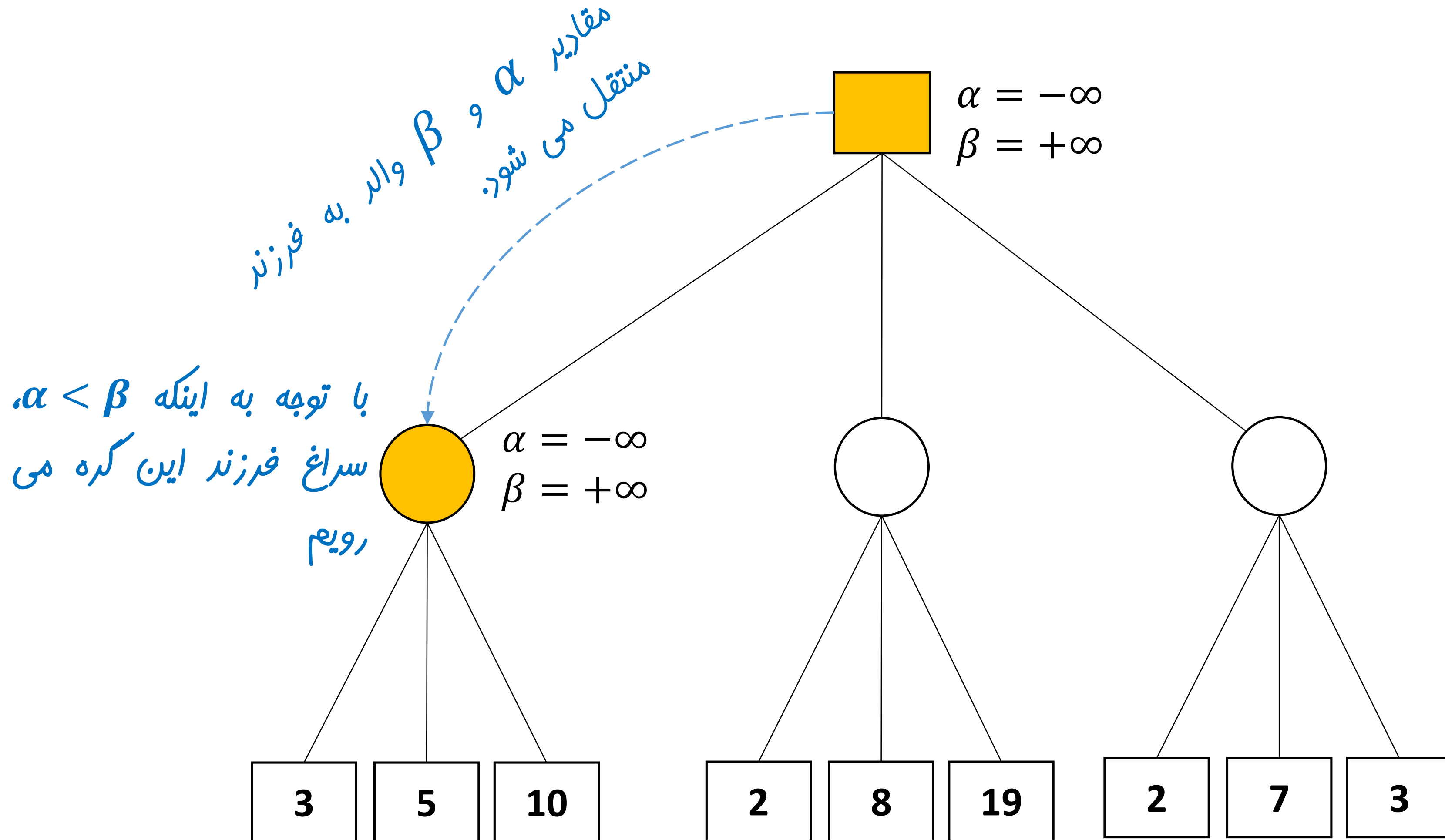
مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:




با توجه به اینکه $\alpha < \beta$، بنابراین
این گره هرس نمی شود.

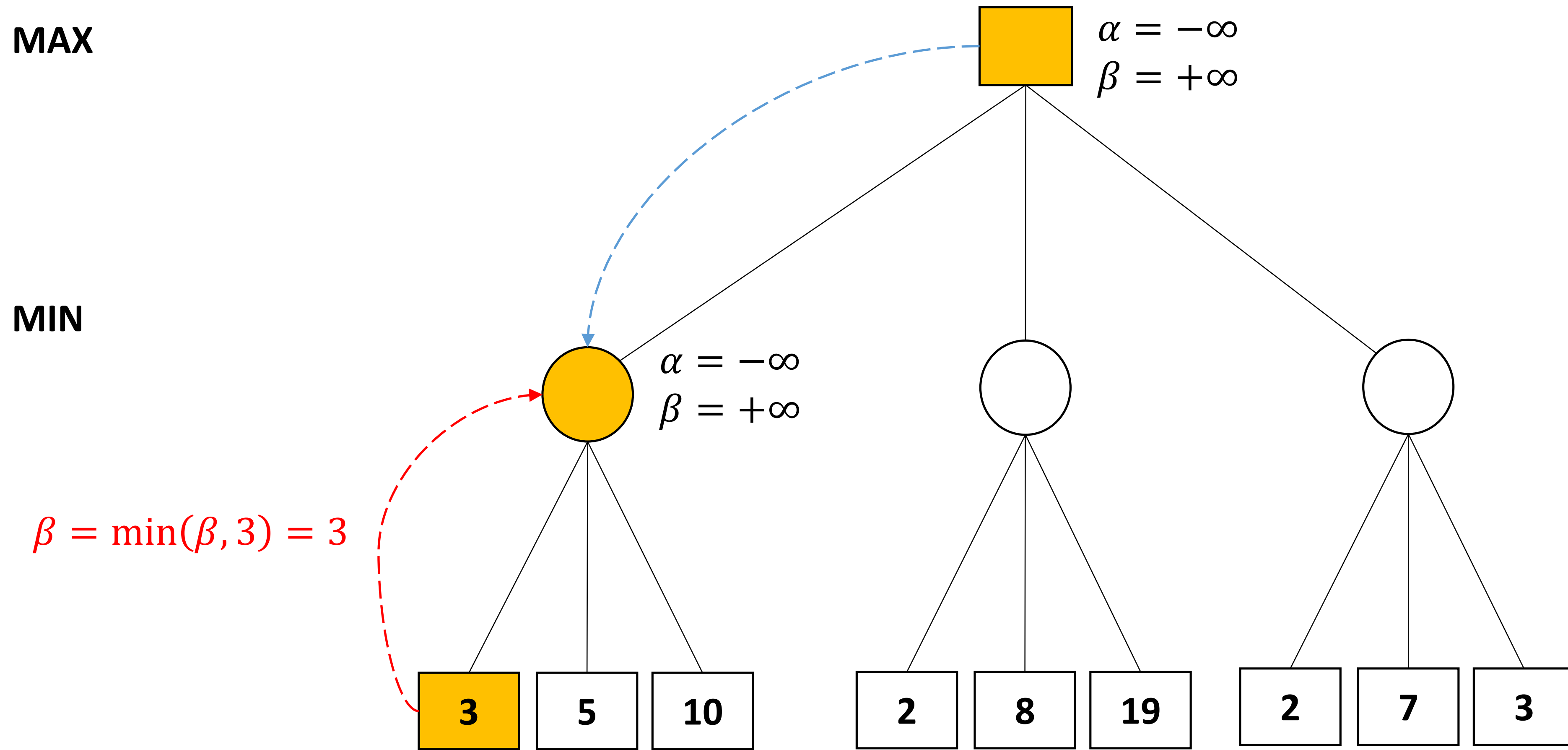
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



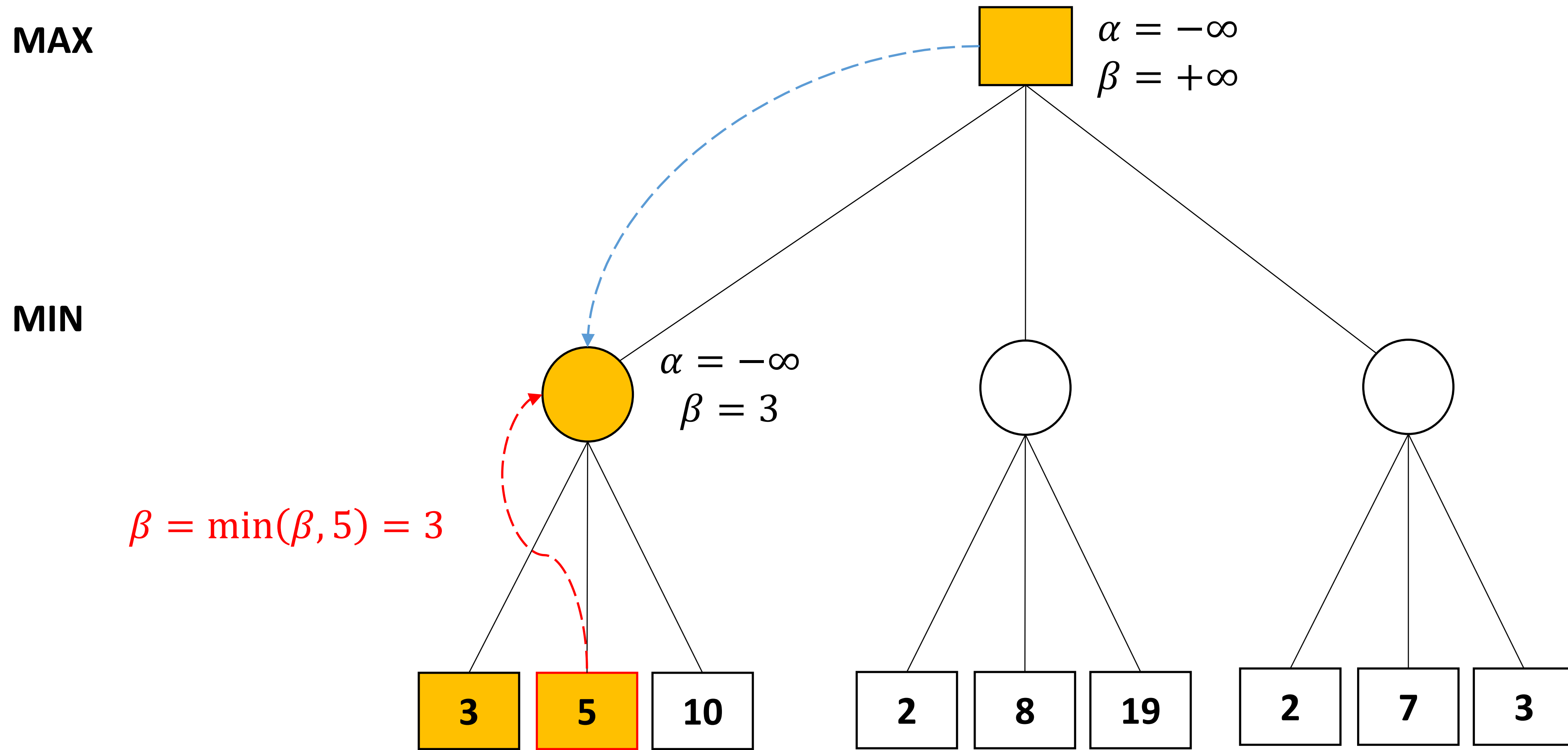
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا–بتا هرس کنید:



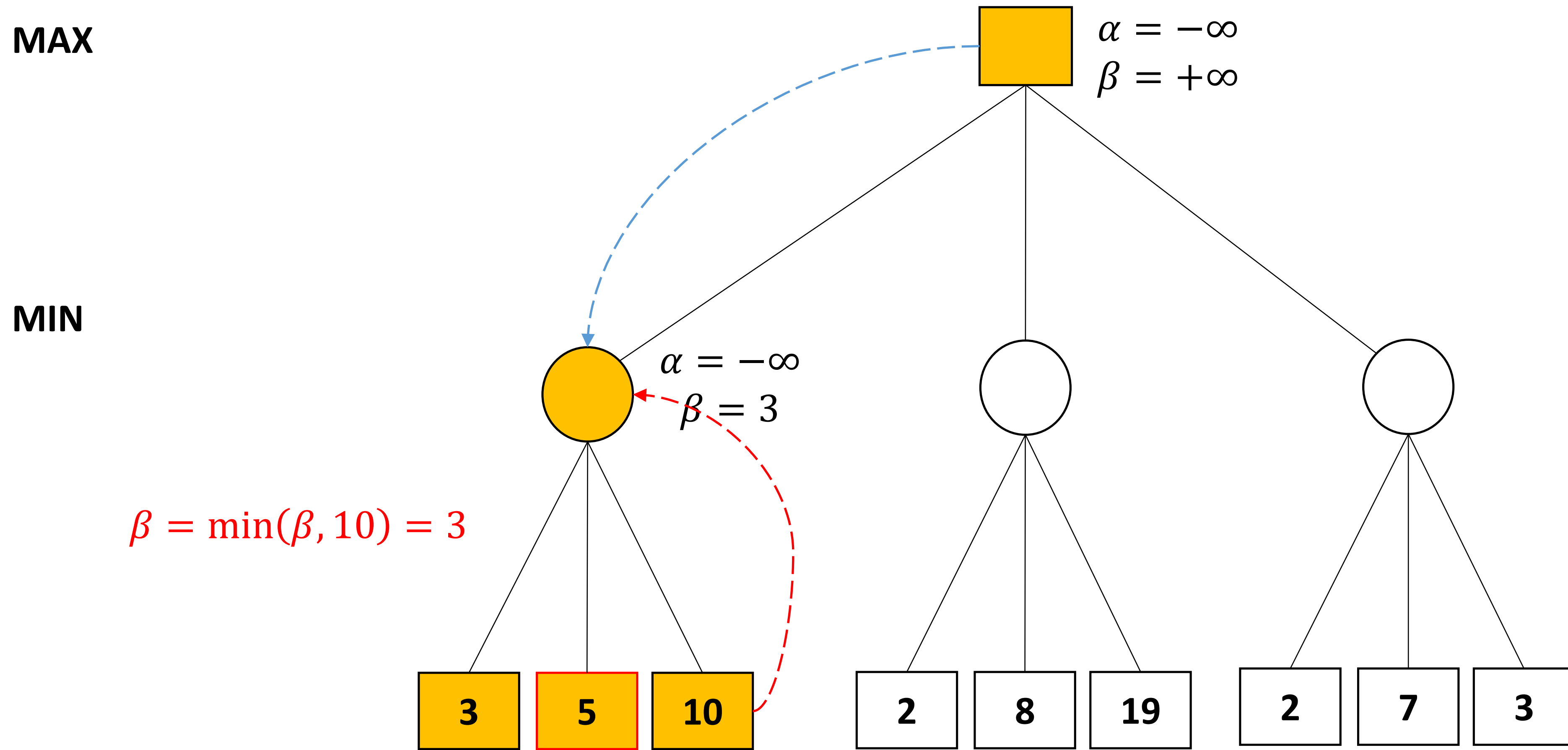
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



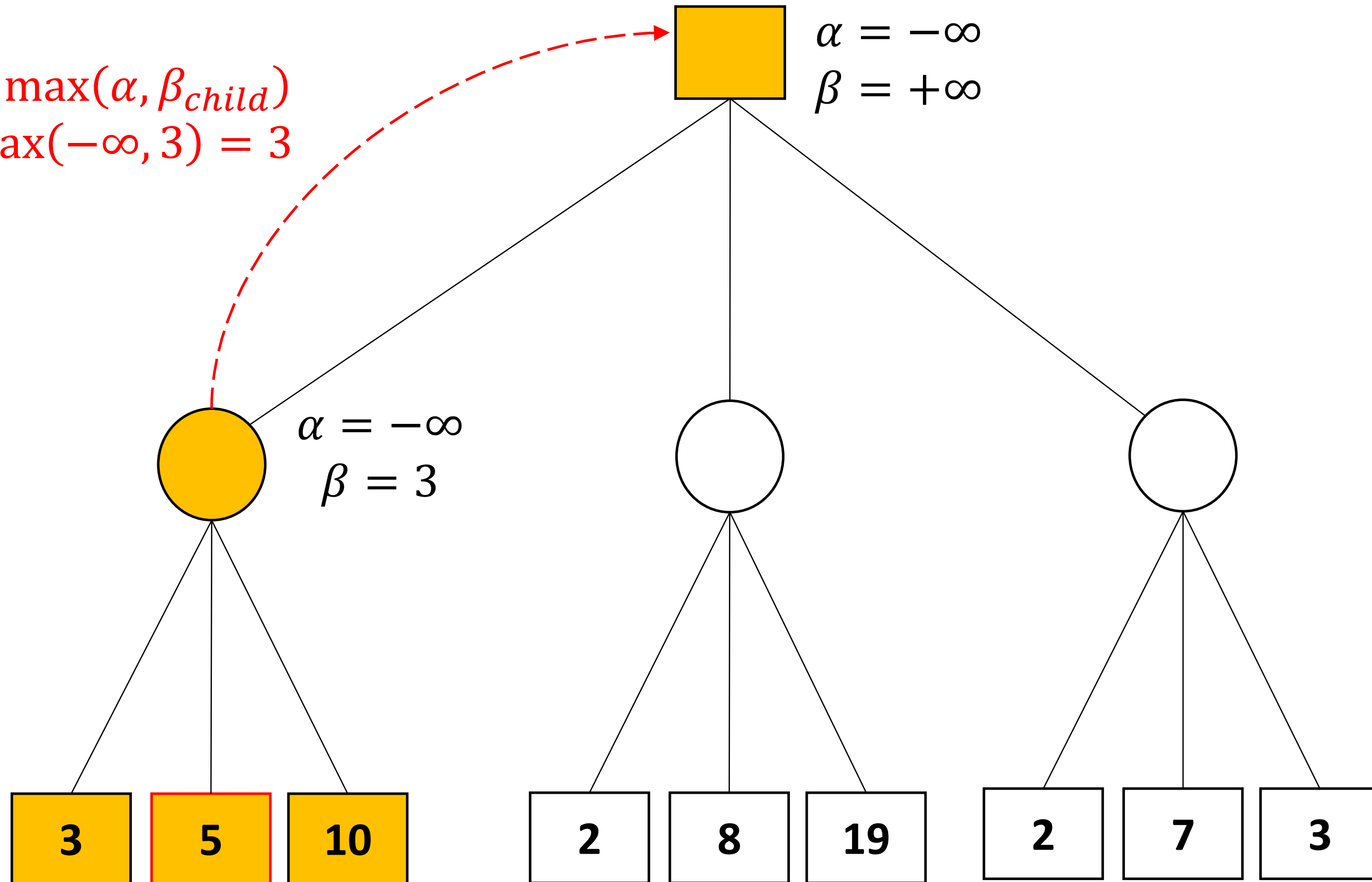
# بهبود الگوریتم MINIMAX: هرس $\alpha-\beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



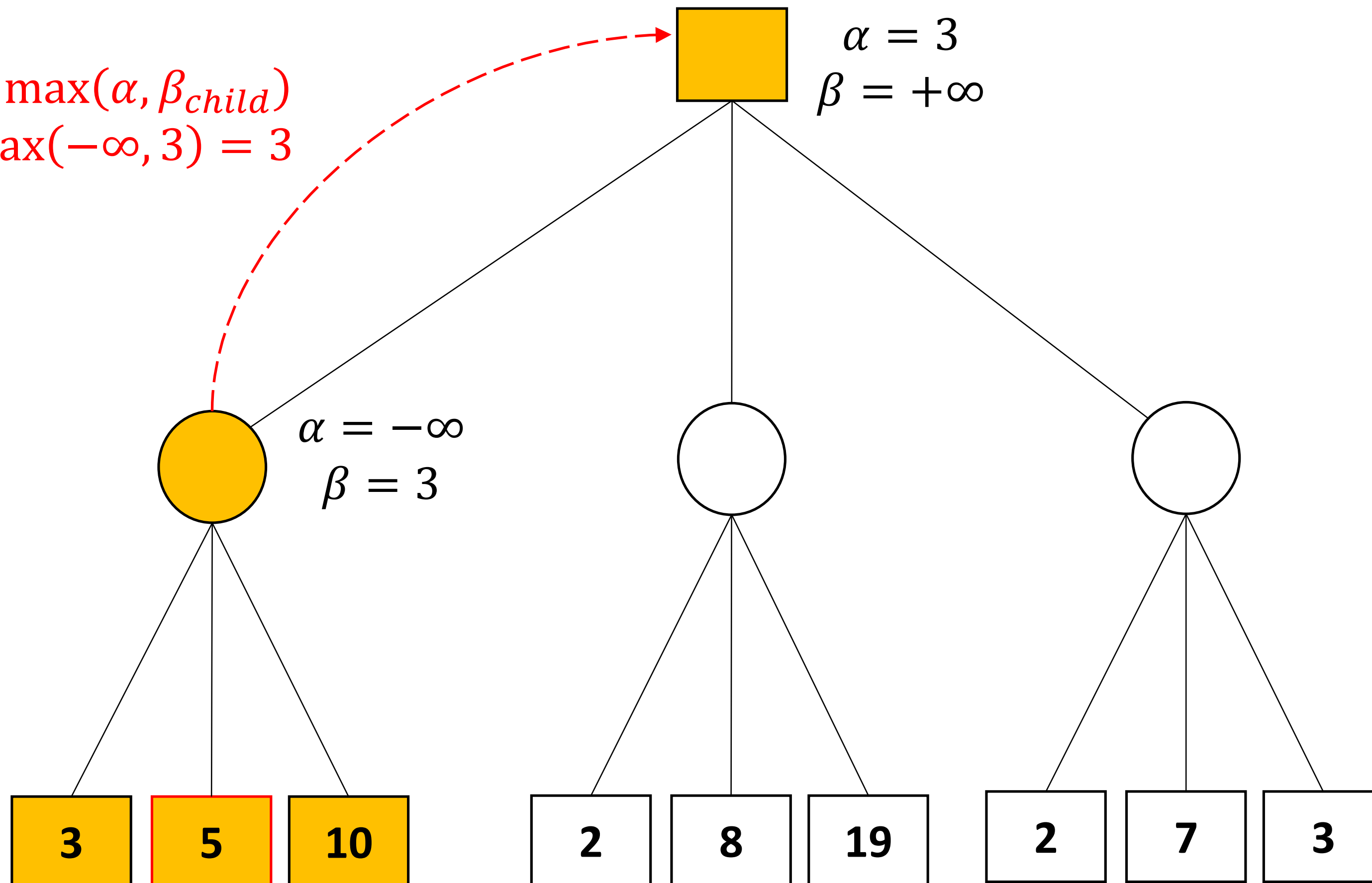
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



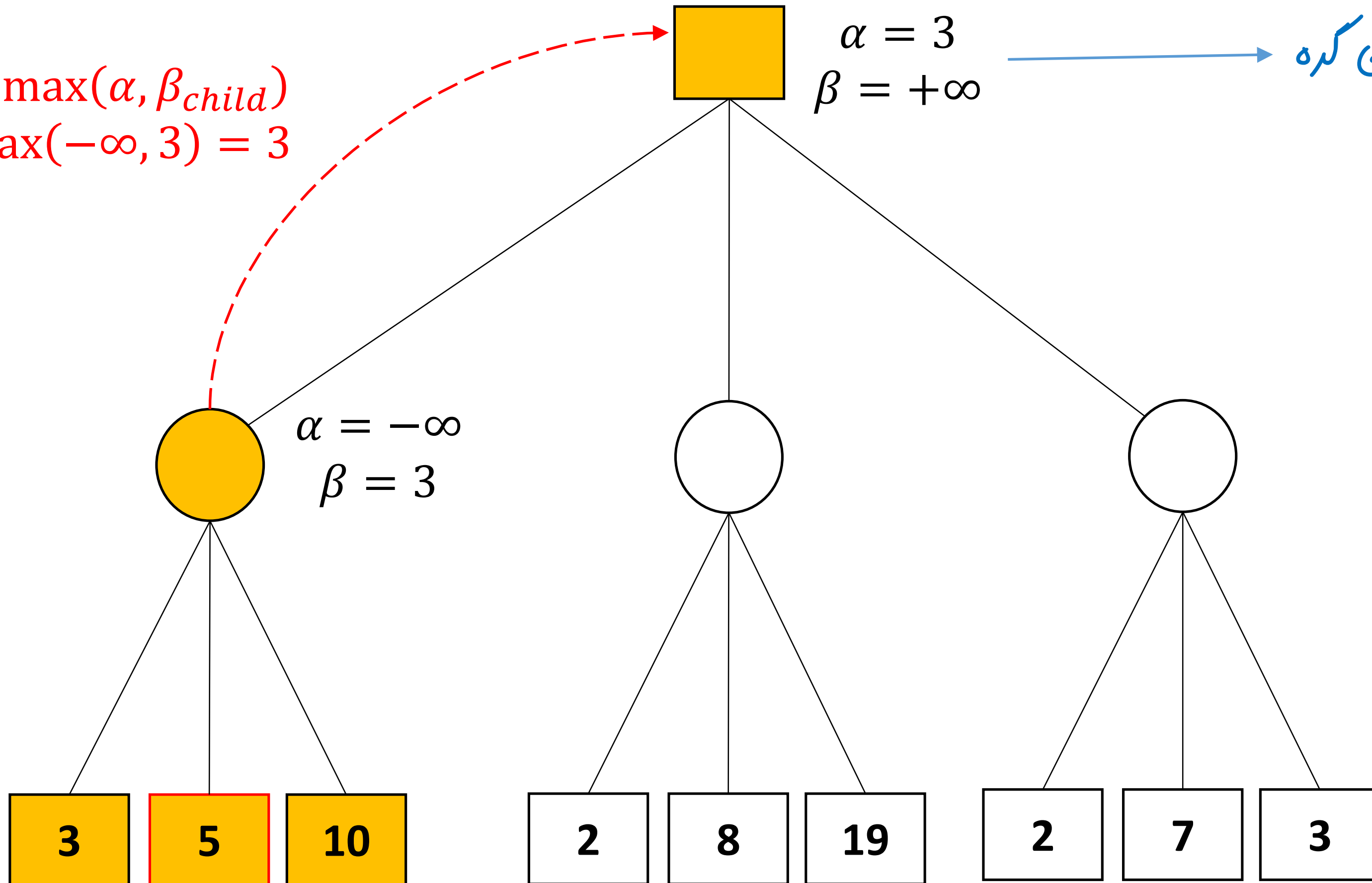
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

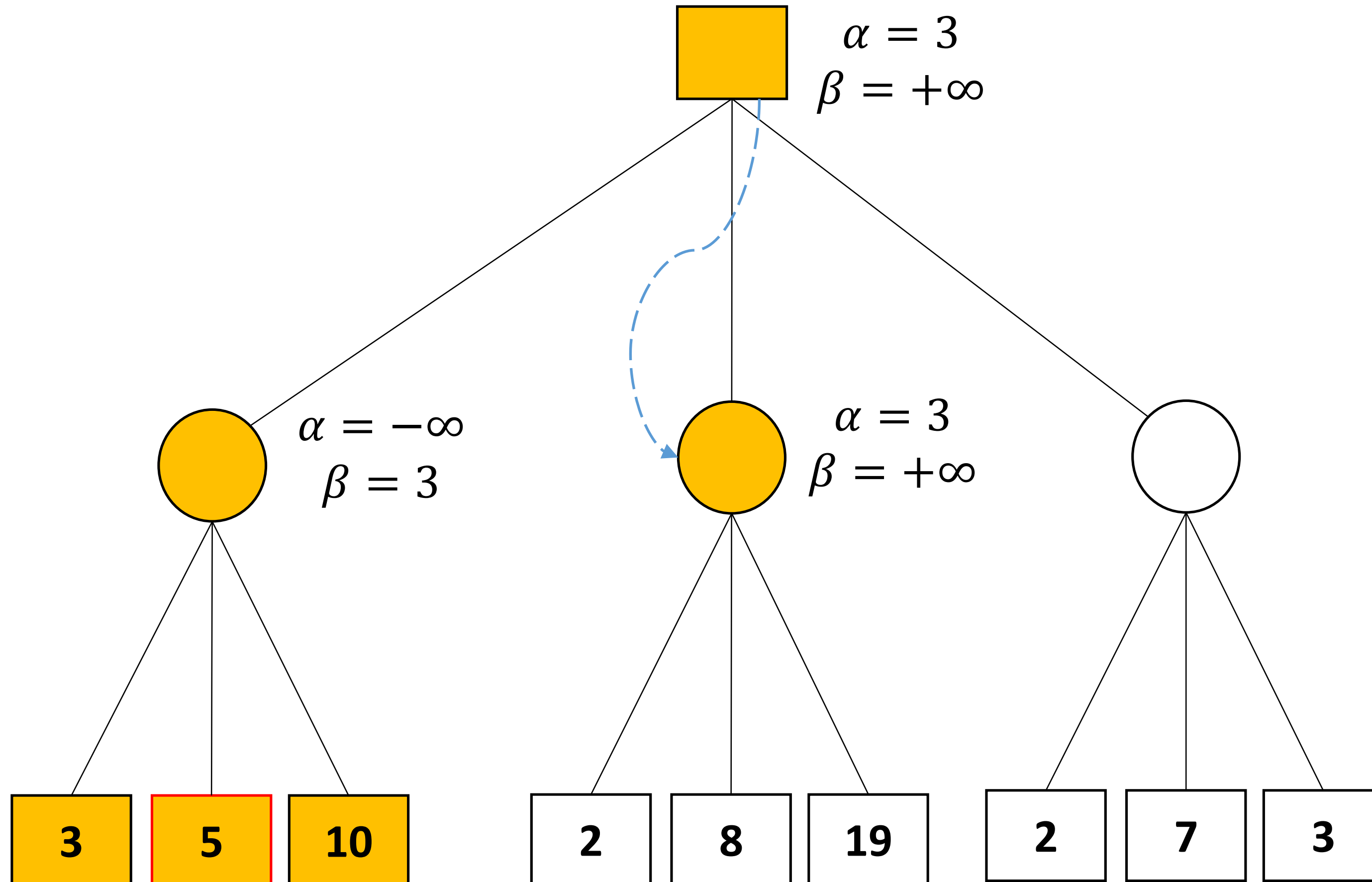
مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:




با توجه به اینکه هنوز $\alpha < \beta$،
بنابراین سراغ فرزند دیگر این گره
می رویم

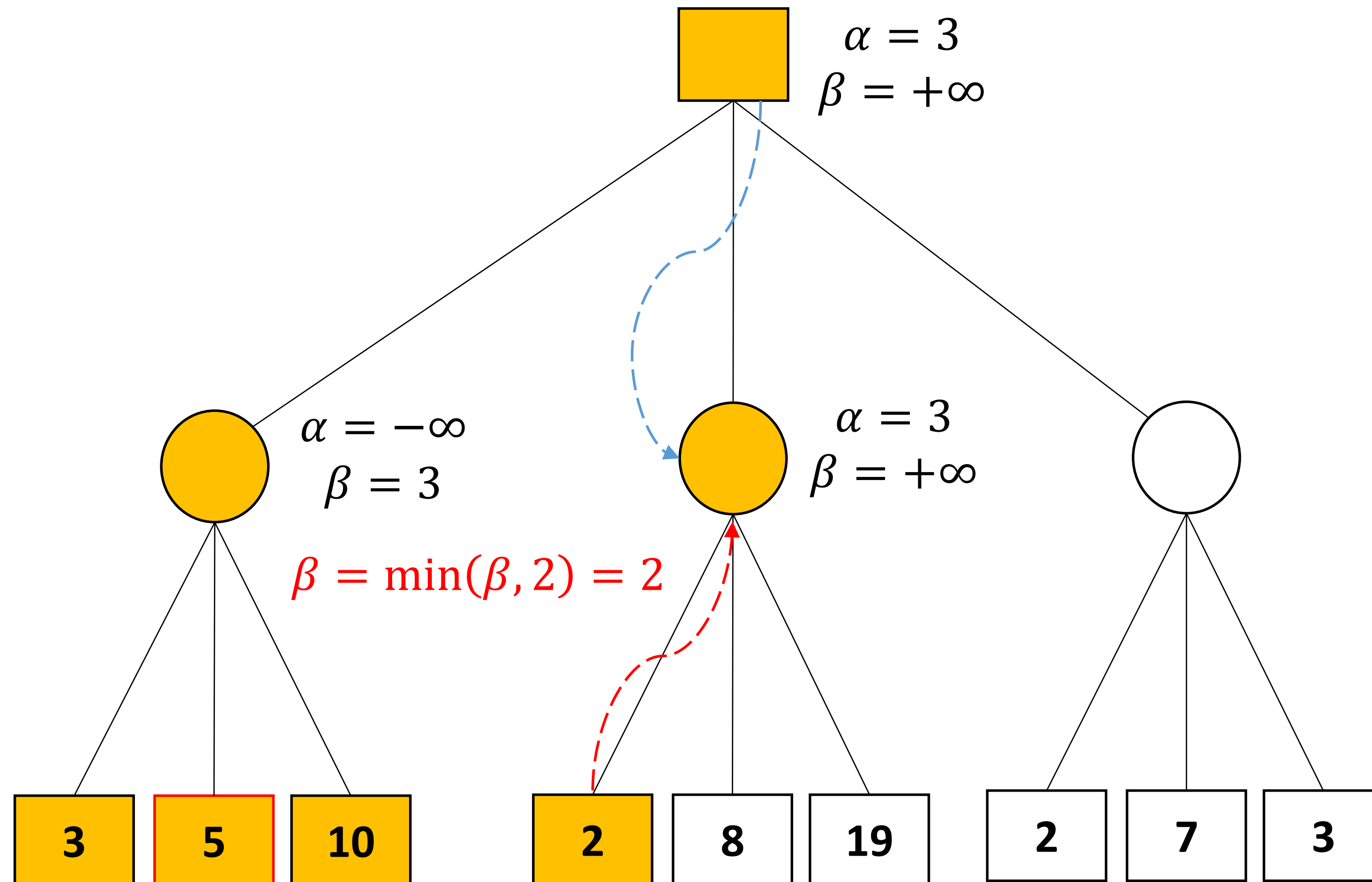
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



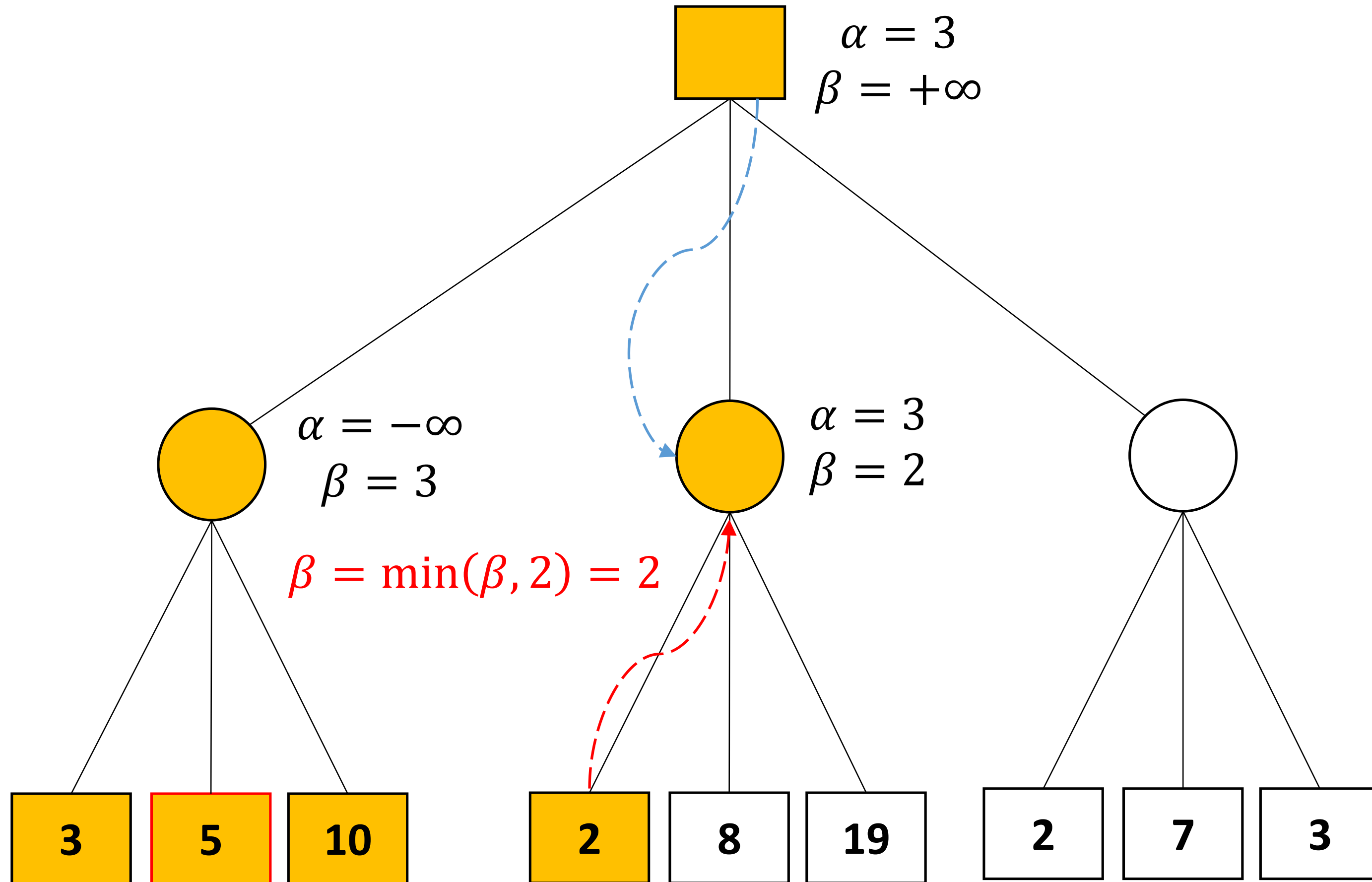
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا–بتا هرس کنید:



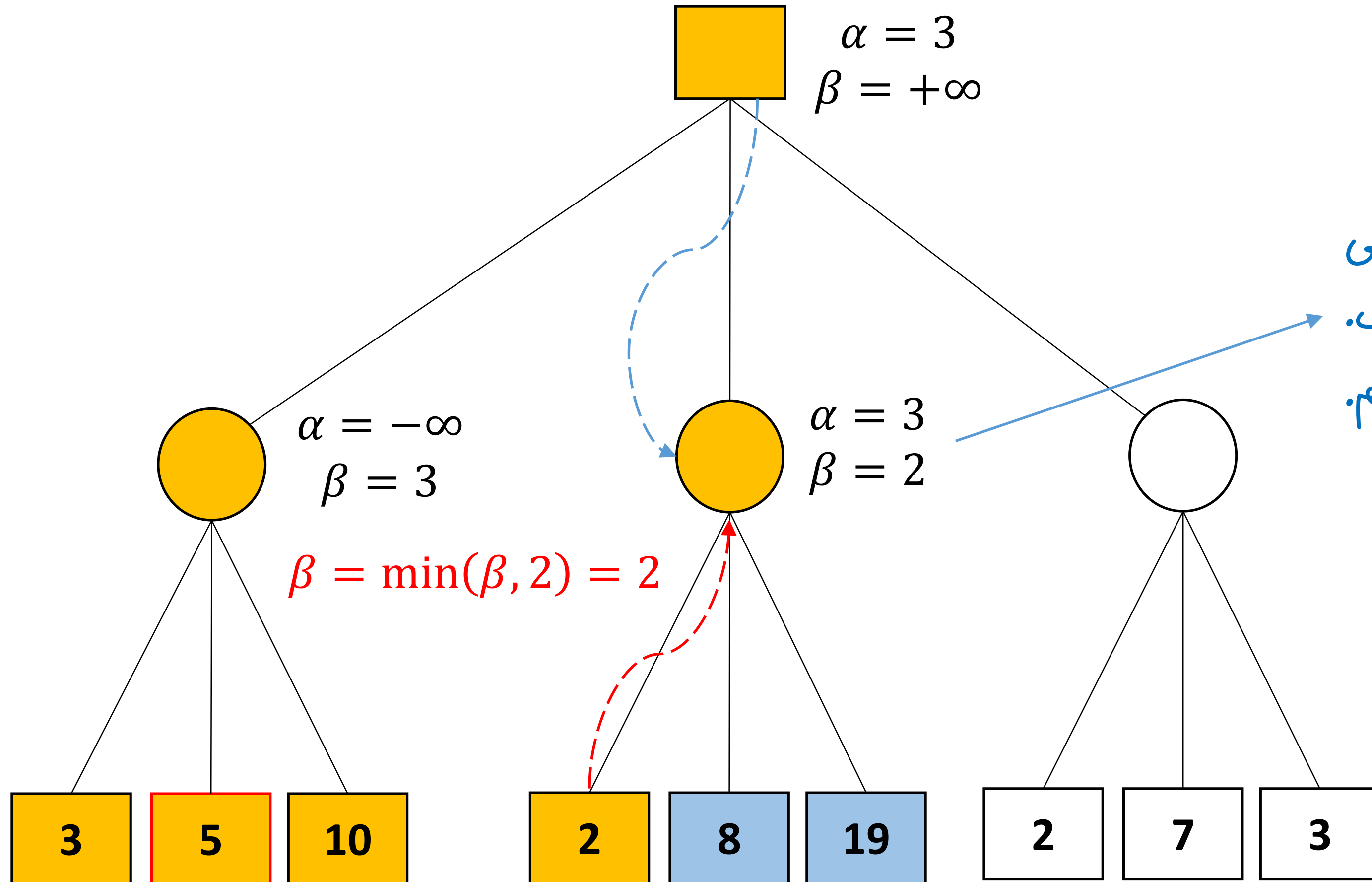
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

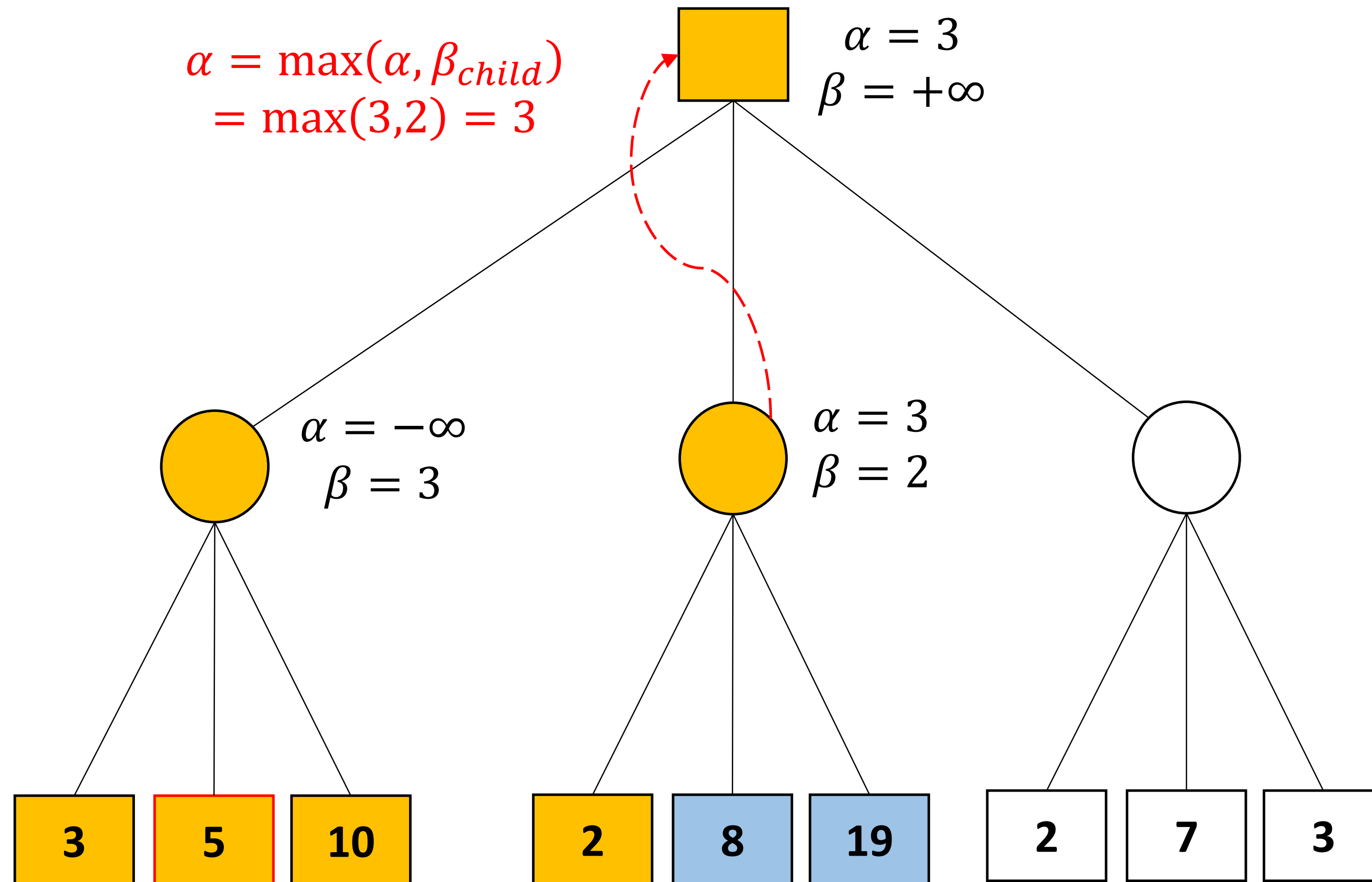
مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:




با توجه به اینکه $\alpha > \beta$، بررسی فرزندان دیگر نیاز نیست. بنابراین، به ریشه عقبگرد می کنیم.

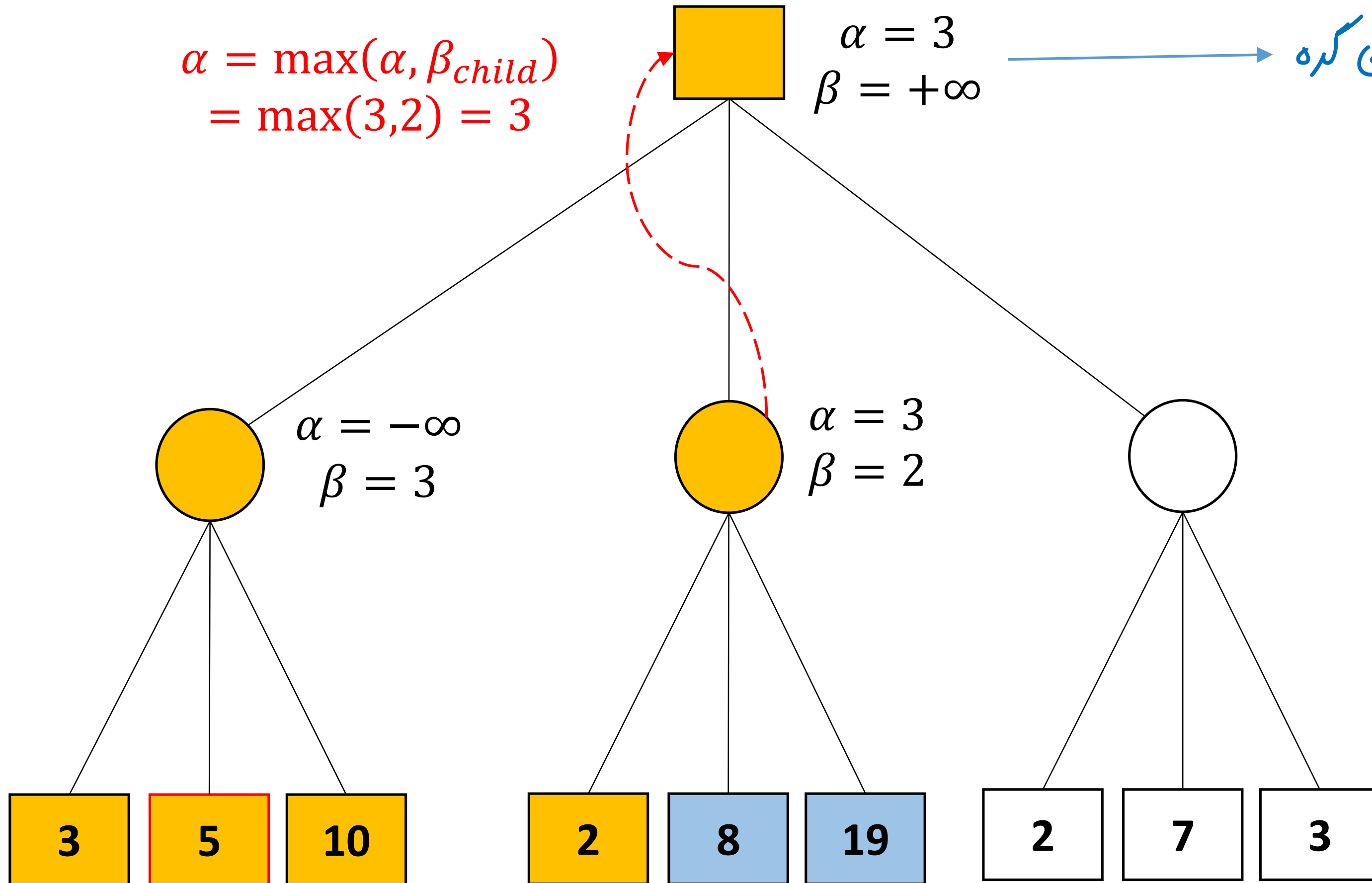
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



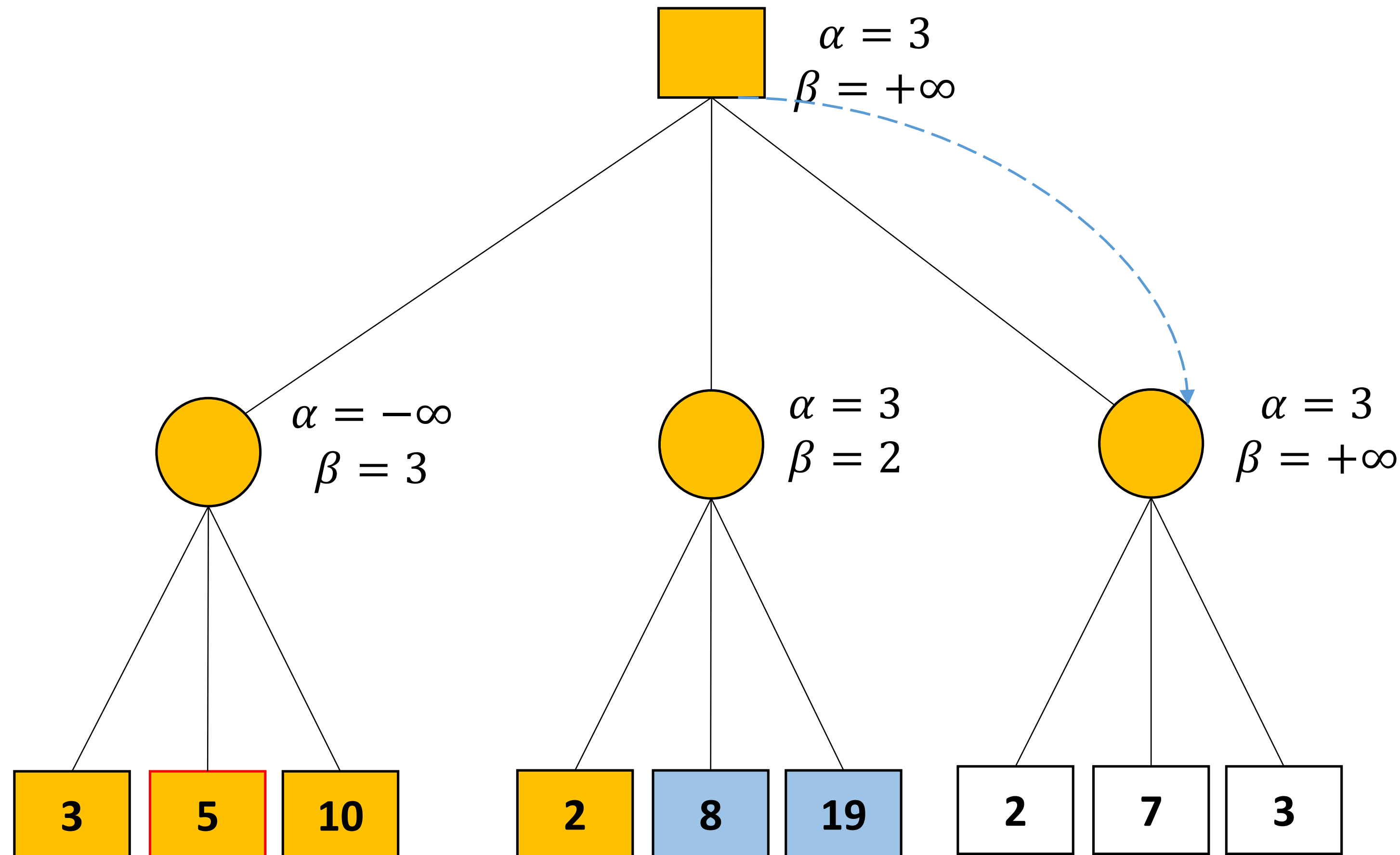
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



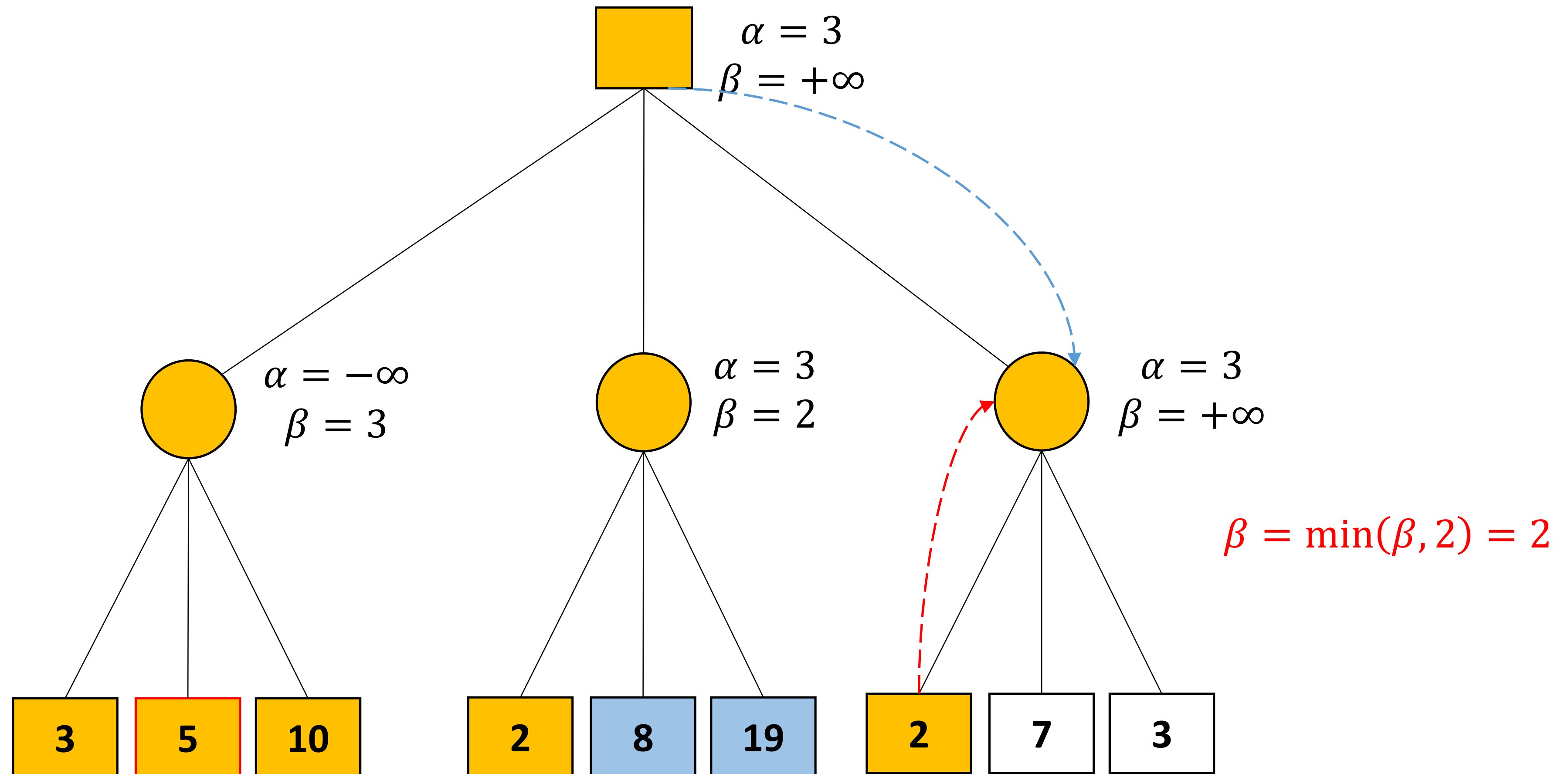
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



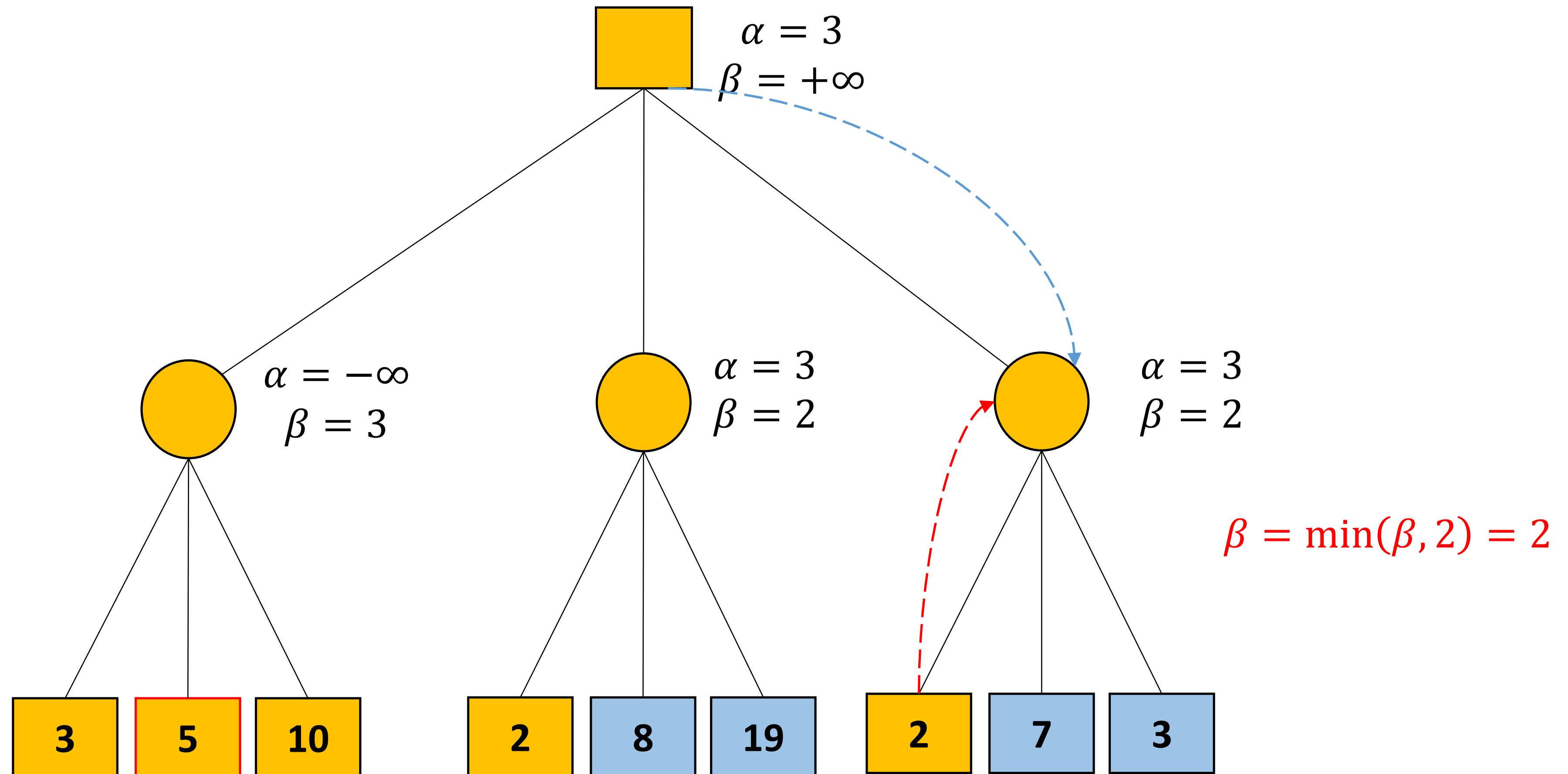
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا–بتا هرس کنید:



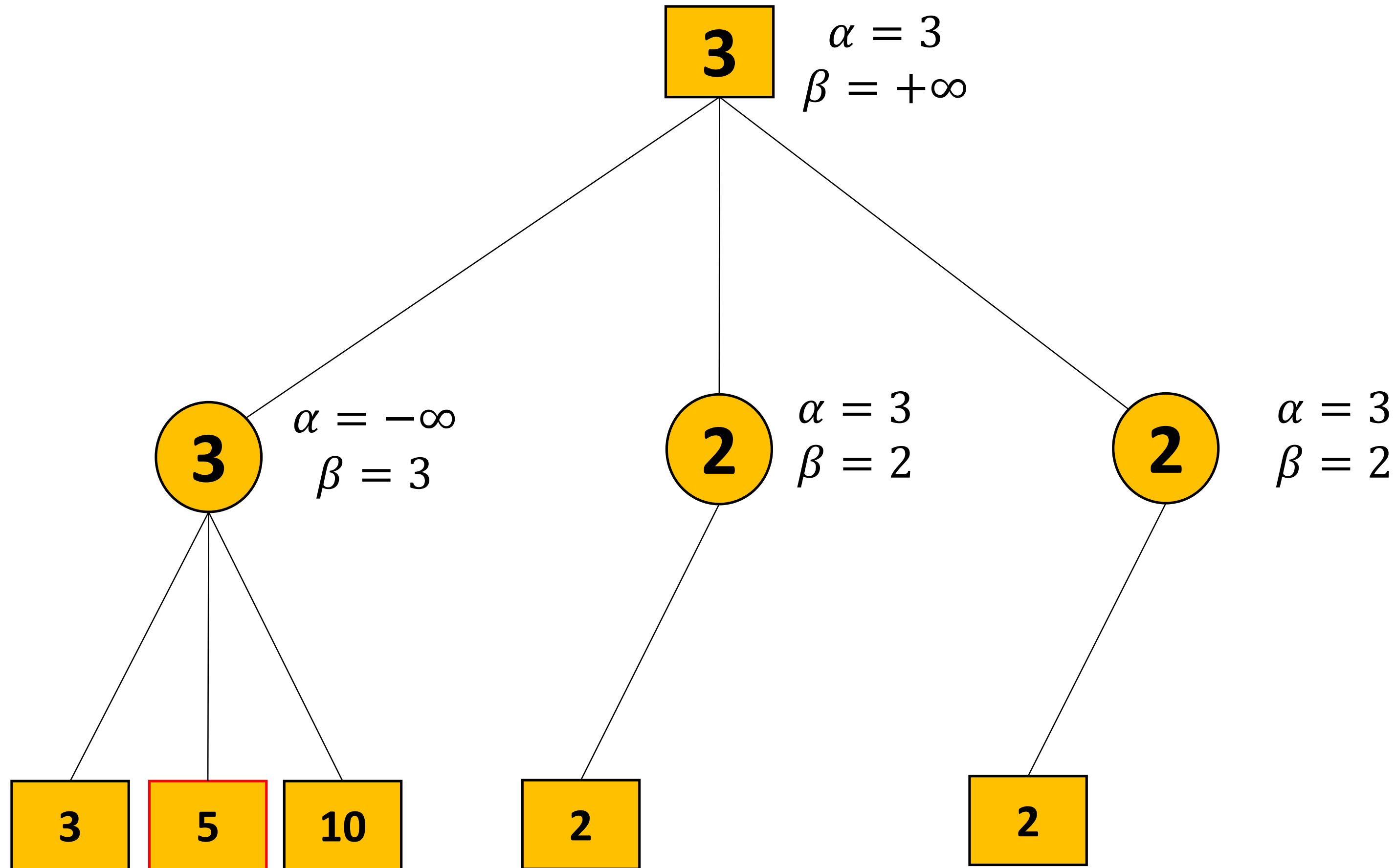
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



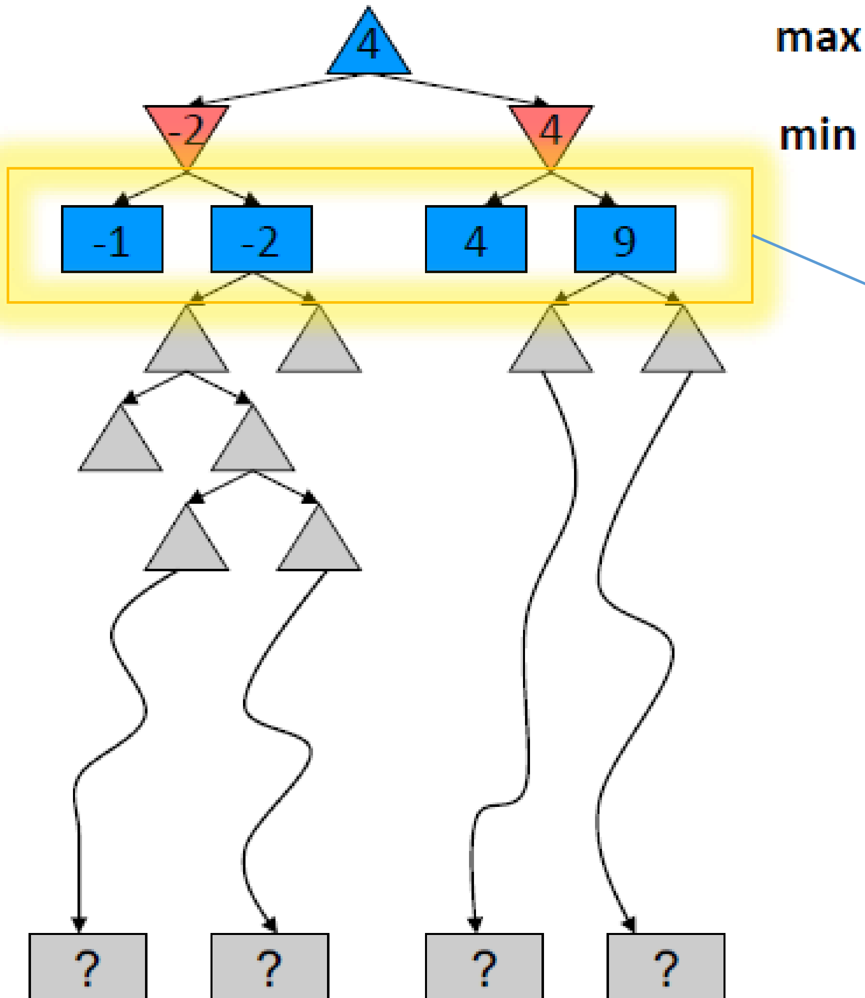
# بهبود الگوریتم MINIMAX: هرس $\alpha - \beta$

مثال: درخت بازی زیر را با استفاده از آلفا-بتا هرس کنید:



# بهبود الگوریتم MINIMAX

اگرچه با استفاده از روش هرس $\alpha - \beta$، بخش بزرگی از درخت جستجو حذف می شود، ولی برای بازی های واقعی، مانند شطرنج، رسیدن به گرههای ریشه (Utility Value ها) عملاً غیرممکن است.




یک عمق را انتخاب می کنیم و از یک تابع ارزیاب برای ارزیابی گرههای این عمق استفاده می کنیم. سپس از الگوریتم minimax استفاده می کنیم.

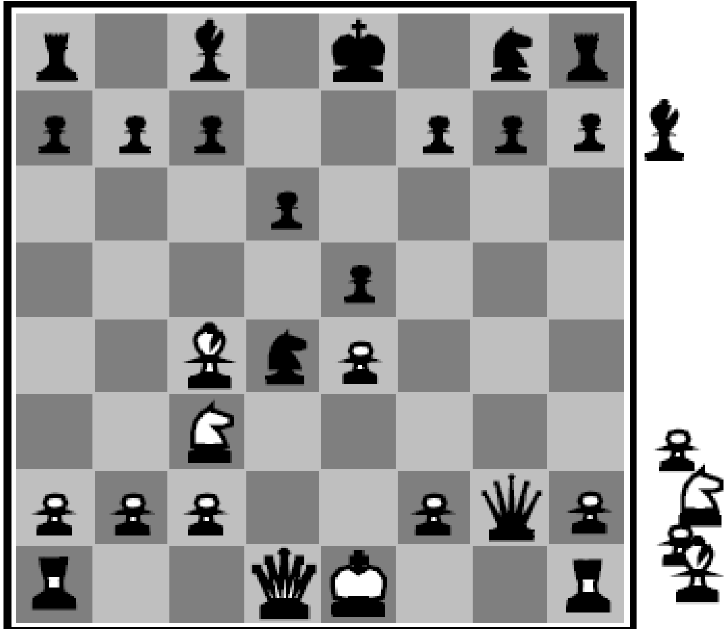
راه حل: جستجو با عمق محدود + استفاده از یک تابع ارزیاب برای گرههای غیر پایانی

# بهبود الگوریتم MINIMAX

- توابع ارزیاب همواره غیرکامل (غیر دقیق) هستند.
- هرچه عمق انتخابی بیشتر باشد، هزینه محاسباتی بیشتر می شود ولی در عوض دقت ارزیابی نیز بیشتر خواهد شد.
- تابع ارزیاب ایده آل تابعی است که مقدار دقیق minimax-value آن گره را محاسبه کند.
- اغلب توابع ارزیاب، برای ارزیابی یک حالت، یک سری ویژگی ها تعریف کرده و آنها را اندازه گیری می کنند.

# بهبود الگوریتم MINIMAX

مثال: ارزیابی یک حالت از بازی شطرنج



$f_1(n)$: تعداد وزیر سیاه – تعداد وزیر سفید

$f_2(n)$: تعداد اسب سیاه – تعداد اسب سفید

$f_3(n)$: تعداد فیل سیاه – تعداد فیل سفید

⋮

$$eval(n) = w_1 f_1(n) + w_2 f_2(n) + w_3 f_3(n) + \cdots$$

$\boldsymbol{w_i}$: میزان اهمیت ویژگی i ام