دانشگاه صنعتی امیرکبیر
دانشکده مهندسی کامپیوتر و فناوری اطلاعات

جستجوی خصمانه

«هوش مصنوعی: یک رهیافت نوین»، فصل ۵

ارائه‌دهنده: سیده فاطمه موسوی

نیم‌سال اول ۱۳۹۹–۱۳۹۸

رئوس مطالب

- محیط‌های چندعاملی
  - بازی‌ها و مسائل جستجو
- معرفی دو الگوریتم معروف بازی‌ها
  - الگوریتم بیشینه کمینه
  - الگوریتم آلفا–بتا
- تصمیمات بی‌درنگ ناقص
- بازی‌های دارای عنصر شانس

محیط‌های چندعاملی

- در محیط‌های چندعاملی هر عامل باید فعالیت سایرعامل‌ها و تأثیر آنها بر روند کار خود را در نظر بگیرد.
  - رفتارهای غیر قابل پیش‌بینی عامل‌های دیگر می‌تواند باعث بروز مقتضیات بسیاری در فرآیند حل مسأله شود.
- رقابتی ← محیط‌های خصمانه (بازی)
  - اهداف عامل‌ها با هم در تضاد هستند
  - هر عامل سعی می‌کند کارایی خود را افزایش دهد
- همکار
  - عامل‌ها اهداف مشترکی دارند.
  - عملی که یک عامل انجام می‌دهد باعث افزایش سودمندی دیگر عامل‌ها می‌شود.

بازی

- در هوش مصنوعی، ”بازی‌ها“ نوع خاصی از مسائل به شمار می‌روند.
- فرضیات اصلی در مورد بازی‌های هوش مصنوعی
  - دو نفره
  - نوبتی
    - هر عامل به نوبت عمل انجام می‌دهد
  - مجموع صفر Zero-Sum
    - اهداف عامل متناقض است مجموع مقادیر سودمندی در پایان بازی صفر یا مقدار ثابتی است.
  - قطعی
  - با اطلاعات کامل Perfect information
    - کاملا مشاهده‌پذیر

بازی به‌عنوان نوعی از مسأله جستجو

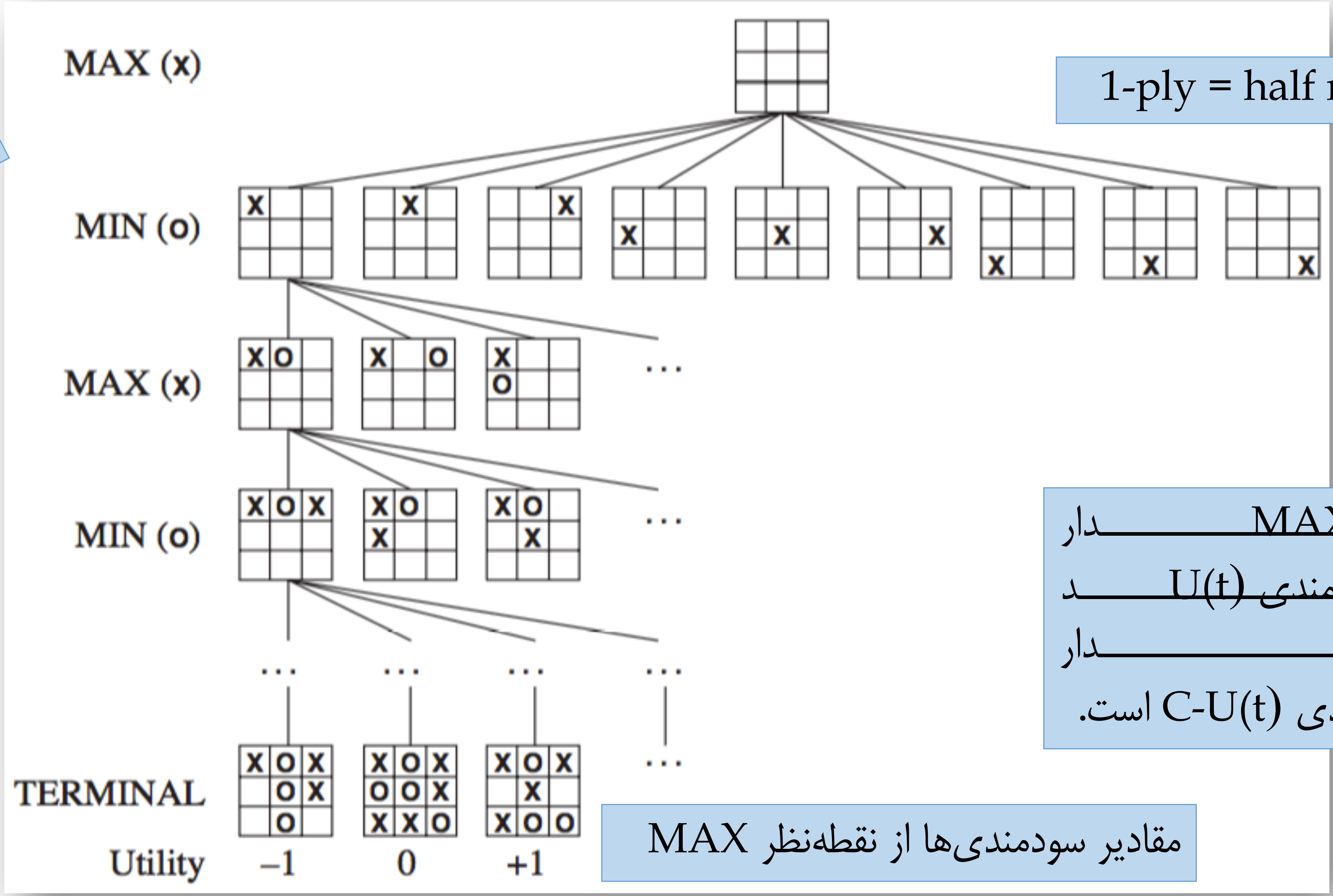
- حالت اولیه s_0: موقعیت اولیه بازی را مشخص می‌کند.
- Player(s): در وضعیت s نوبت کدام بازیکن است.
- Actions(s): مجموعه اعمال قانونی در وضعیت s را برمی‌گرداند.
- Result(s,a): مدل انتقال، نتیجه یک حرکت را تعریف می‌کند.
- Terminal-Test(s): آزمون پایانی هنگامی که بازی تمام شده باشد درست و در غیر این‌صورت غلط برمی‌گرداند.
- Uitlity(s,p): مقدار سودمندی بازیکن p در حالت پایانی s چقدر است.
  - بازی مجموع صفر (مجموع ثابت): مجموع مقدار سومندی تمام بازیکنان در حالت پایانی s برابر با صفر یا یک مقدار ثابت است.

بازی به‌عنوان نوعی از مسأله جستجو ...

- درخت بازی = حالت اولیه + تابع اقدامات + تابع نتیجه
  - درختی که گره‌ها در آن وضعیت‌های بازی هستند و یال‌ها حرکات
  - چون رقیب غیر قابل پیش‌بینی است باید یک حرکت برای هر پاسخ ممکن از طرف رقیب مشخص نمود
    - مانند جستجوی AND-OR
- در ادامه فرض می‌کنیم
  - دو بازیکن MAX و MIN داریم که شروع‌کننده‌ی بازی MAX است.
  - هدف ما یافتن بهترین عملی است که MAX می‌تواند انجام دهد تا بیشترین سودمندی را به‌دست آورد.

درخت بازی (دوز tic-tac-toe)

دو بازیکن MAX و MIN
که الان MAX به دنبال
بهترین حرکت خود است



1-ply = half move

ر MAX دار
ودمندی U(t) د
Min دار
سودمندی C-U(t) است.

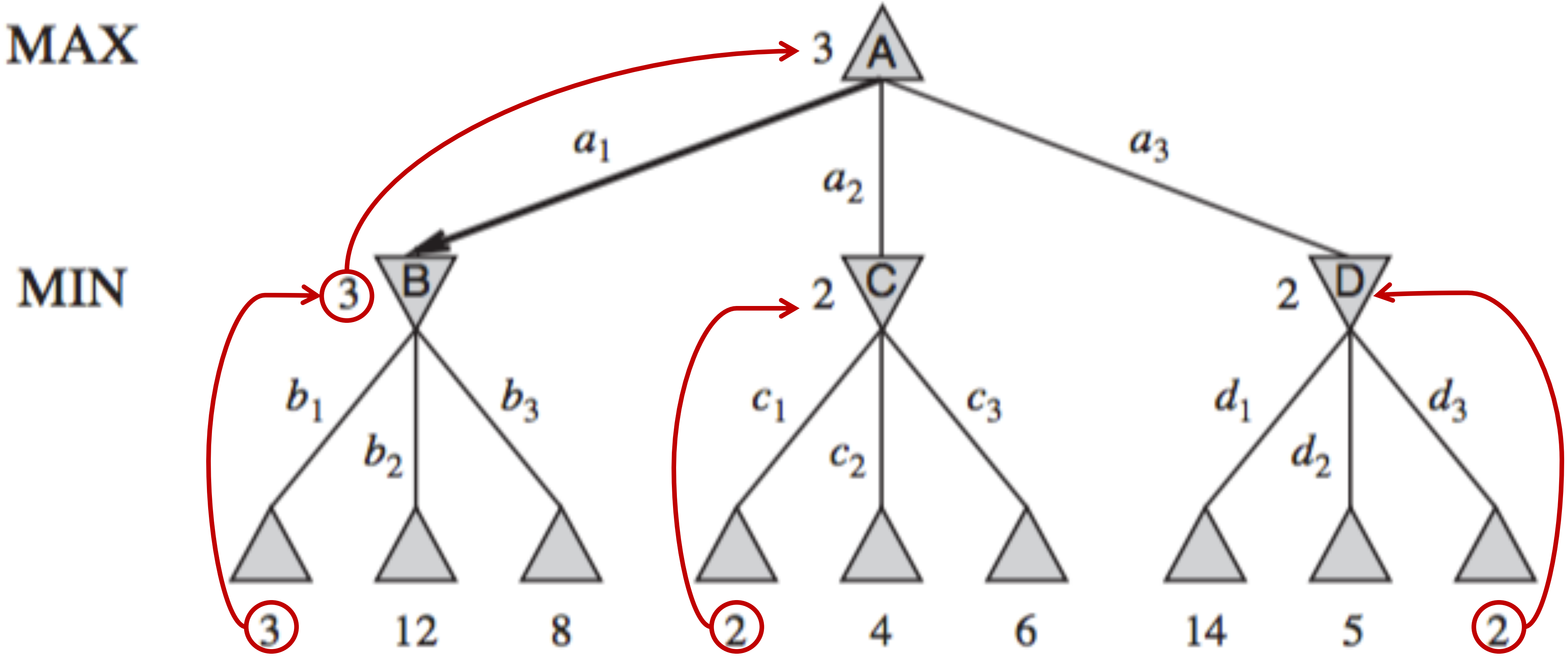
مقادیر سودمندی‌ها از نقطه‌نظر MAX

استراتژی کمینه‌بیشینه (MINIMAX)

- با داشتن درخت بازی، استراتژی بهینه را می‌توان با در نظر گرفتن مقدار minimax گره‌ها تعیین نمود.
  - تابع MINIMAX(s) بهترین نتیجه‌ی بودن در وضعیت s را مشخص می‌کند. با فرض آن‌که هر دو بازیکن از گره شروع تا پایان بهینه بازی کنند.
  - MAX به‌دنبال بیشینه کردن مقدار سودمندی خود و MIN به‌دنبال کمینه کردن مقدار سودمندی حریف است.

$$
\textsc{Minimax}(s) =
\begin{cases}
\textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\
\max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\
\min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min}
\end{cases}
$$

کمینه‌بیشینه (مثال ۱)



MINIMAX بدترین نتیجه برای MAX را بیشینه می‌کند چون فکر می‌کند همیشه MIN می‌خواهد بهینه عمل کند.

الگوریتم کمینه‌بیشینه

function MINIMAX-DECISION(*state*) **returns** *an action*
  return arg max$_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(*s*, *a*)))
  return v

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(*s*, *a*)))
  return v

می‌توان با این قطعه
کد جایگزین نمود

$v \leftarrow$ MAX-VALUE(state)
return the action in successors(state) with value v

در روند اجرای الگوریتم هر جا نود MAX را دیدیم علامت ∞- و هرجا نود MIN را دیدیم علامت ∞+ قرار می‌دهیم.

ویژگی‌های الگوریتم بیشینه‌کمینه

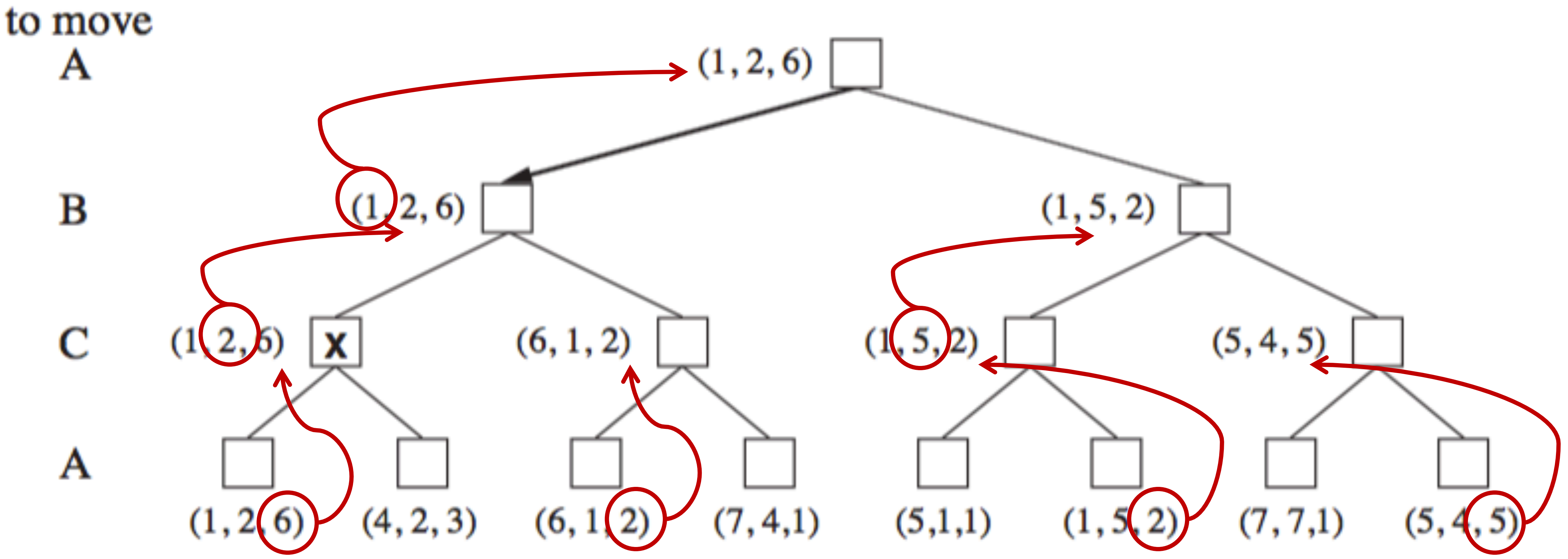
بصورت عمقی پیـاده‌سازی شده است.

- کامل بودن؟ بله
  - هنگامی‌که درخت متناهی باشد و حافظه به اندازه کافی موجود باشد.
- بهینه بودن؟ بله
  - بهترین حرکت در مقابل یک بازیکن حرفه‌ای را انتخاب می‌کند.
  - اگر حریف به‌طور بهینه بازی نکند چه پیش خواهد آمد؟
- پیچیدگی زمانی؟ نمایی $O(b^m)$
  - راه حل دقیق برای بازی‌های واقعی کاملا غیر قابل دسترس است.
  - برای مثال در شطرنج $b \approx 35, m \approx 100$
- پیچیدگی فضایی؟ خطی $O(bm)$

بسط ایده‌ی بیشینه‌کمینه به بازی‌های چند نفره

- مقادیر اختصاص داده شده به هر گره را با یک بردار سودمندی با سایز تعداد بازیکنان جایگزین می‌کنیم.
  - اگر سه بازیکن A، B و C داشته باشیم بردار سودمندی برابر با $<v_A,v_B,v_C>$ خواهد بود.
- برای حالات پایانی، این بردار حاوی مقادیر سودمندی آن حالت از نظر هر بازیکن است.
- بردار سودمندی برای هر گره برابر با برداری است که دارای مقدار سودمندی بیشتر برای بازیکنی باشد که در آن گره دارای حق انتخاب است.
- در بازی‌های چند نفره ممکن است بین بازیکن‌ها اتحاد و یا همکاری بوجود آید.
  - اتحاد: حمله دو بازیکن ضعیف به بازیکن قوی‌تر
  - همکاری: اگر ۱۰۰ بیشترین سودمندی ممکن باشد که تنها در یک حالت پایانه <۱۰۰، ۱۰۰> اتفاق می افتد بازیکن‌ها برای رسیدن به این وضعیت به‌طور خودکار همکاری می‌کنند.

بازی‌های چند نفره – مثال



هرس آلفا–بتا

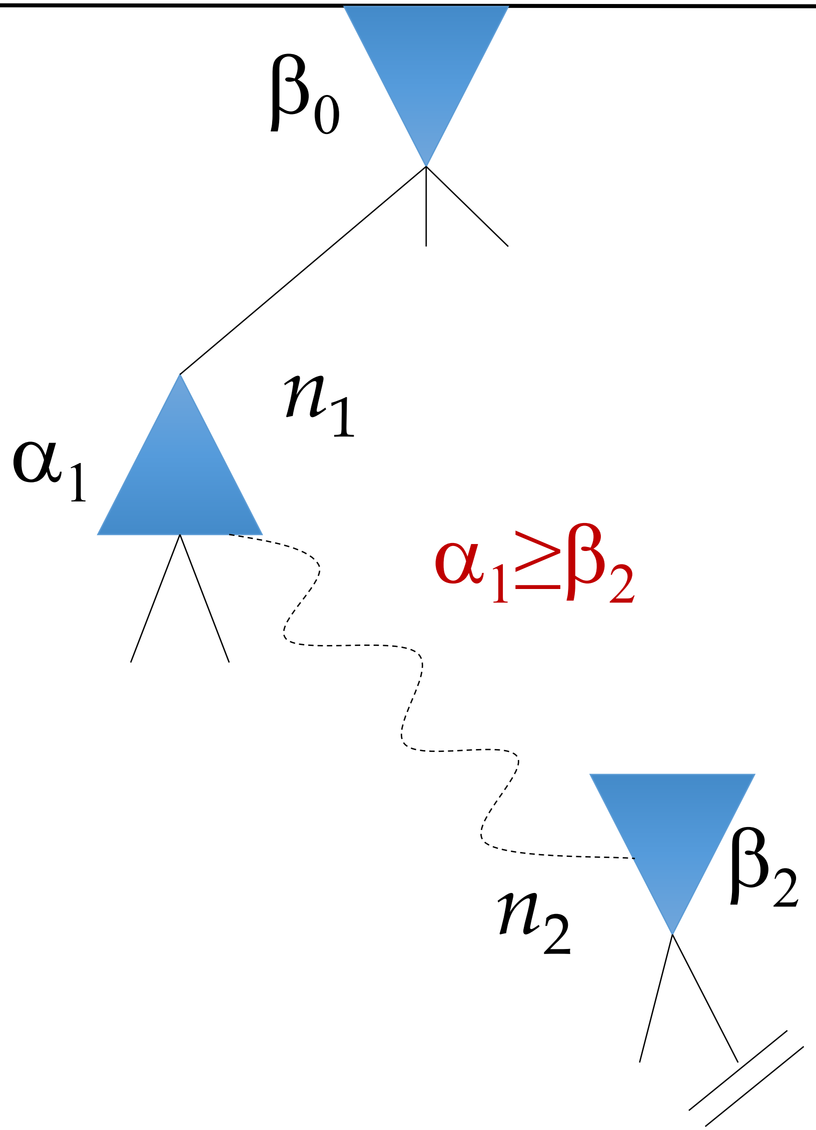
- مشکل الگوریتم بیشینه‌کمینه این است که تمام گره‌های درخت را بررسی می‌کند.
  - این تعداد برحسب تعداد حرکات نمایی است درنتیجه زمان زیادی می‌برد.
  - به هیچ طریقی نمی‌توان رابطه‌ی نمایی را از بین برد اما می‌توان با هرس تعداد حالات بررسی را تقریباً به نصف کاهش داد.
- ایده هرس کردن
  - عدم بررسی برخی شاخه‌ها و افزایش سرعت در تصمیم‌گیری
- هرس آلفا–بتا که به یک درخت بیشینه‌کمینه استاندارد اعمال می‌شود، همان جواب الگوریتم بیشینه‌کمینه را برمی‌گرداند با این تفاوت که در این روش، شاخه‌هایی که در تصمیم‌گیری نهایی تأثیری ندارند، هرس می‌شود.

حقایق هرس آلفا–بتا

دو حقیقت زیر را می‌توان به‌راحتی از روش MINIMAX متوجه شد.

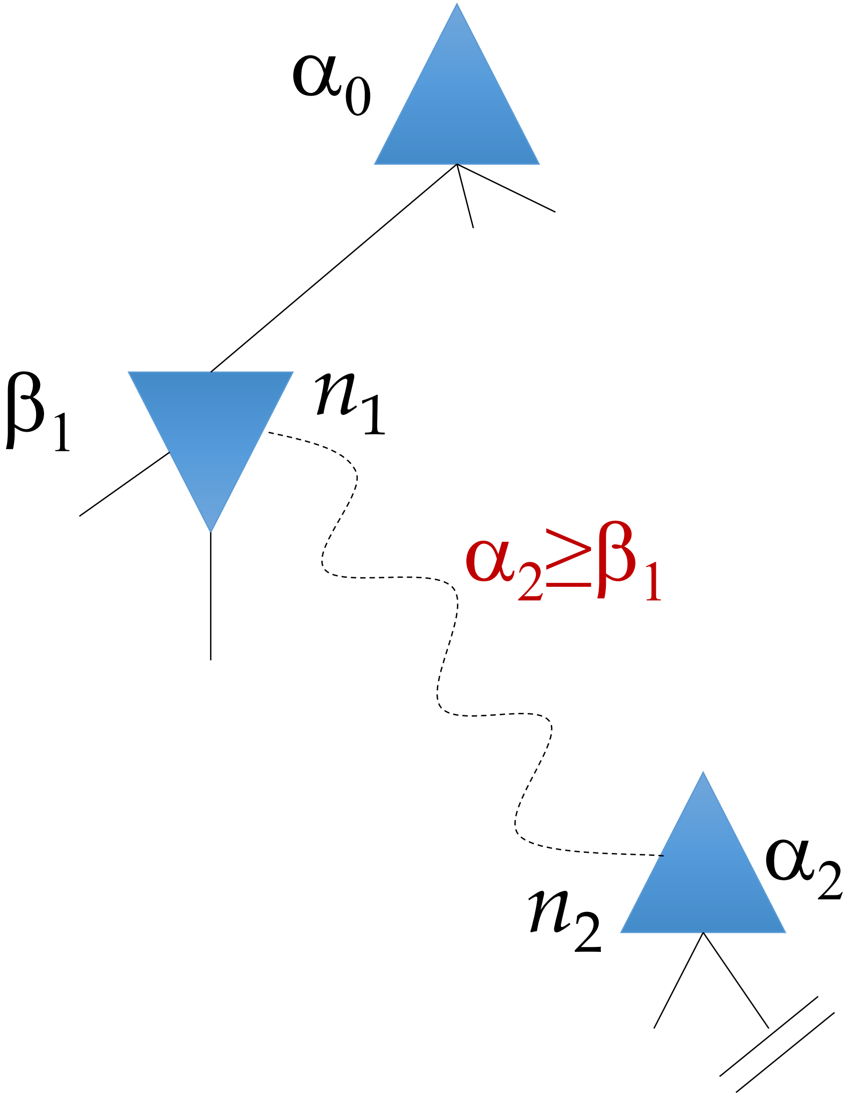
- مقادیر آلفای گره‌های MAX هیچ‌گاه کاهش نمی‌یابند.
  - به گره‌های MAX مقادیر موقت آلفا نسبت داده می‌شود که این مقادیر با دیدن هر یک از فرزندانش همیشه باید در حال افزایش باشد نه کاهش.
  - مقدار اولیه آلفا = -∞
- مقادیر بتای گره‌های MIN هیچ‌گاه افزایش نمی‌یابند.
  - به گره‌های MIN مقادیر موقت بتا نسبت داده می‌شود که این مقادیر با دیدن هر یک از فرزندانش همیشه باید در حال کاهش باشد نه افزایش.
  - مقدار اولیه بتا = +∞

قانون اول – هرس آلفا



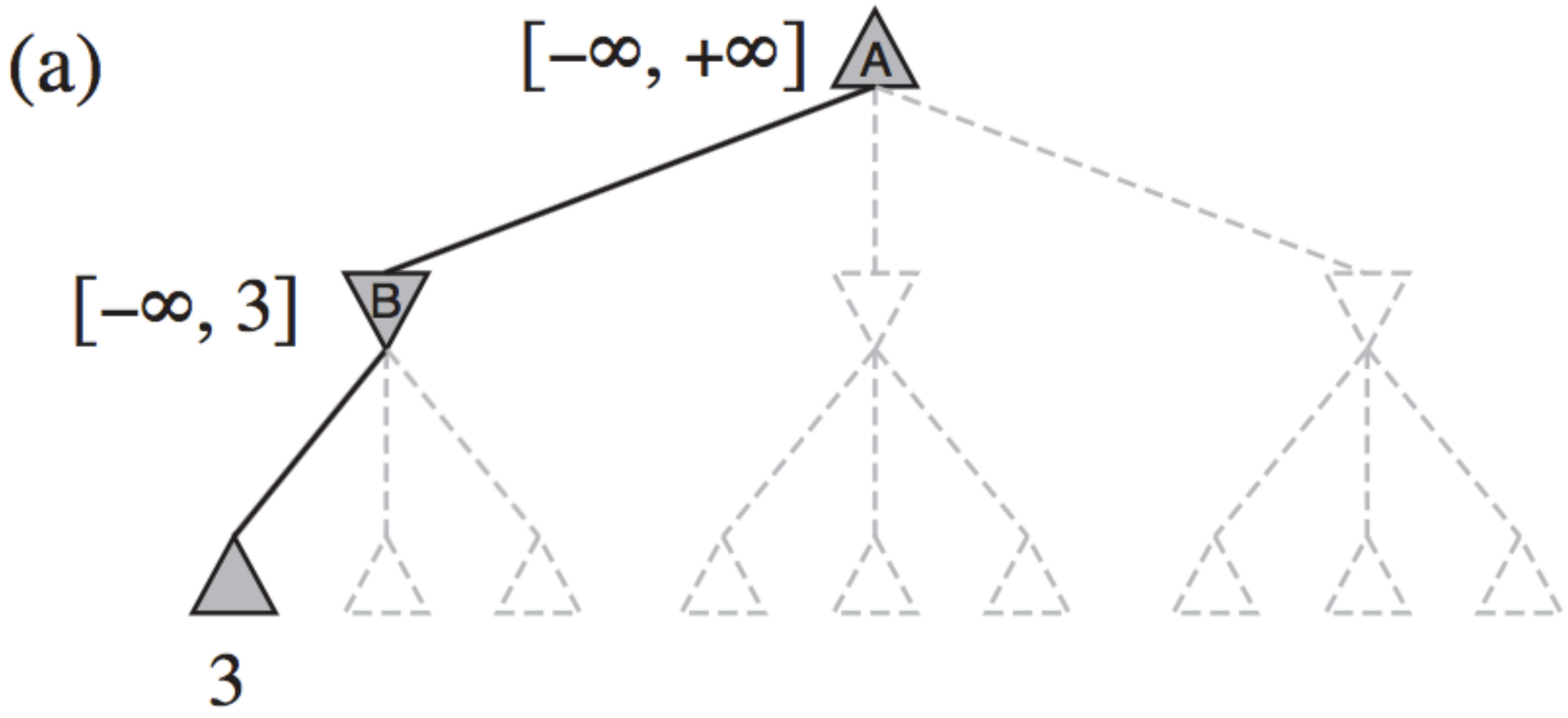
- عمل جستجو تحت گره MIN که مقدار بتای آن کوچک‌تر یا مساوی مقدار آلفای هر گره MAX اجداد آن گره است، قطع می‌شود.
- در این حالت، مقداری که تا به‌حال به گره‌ی MIN نسبت داده شده به‌عنوان مقدار نهایی بتای آن گره انتخاب خواهد شد.

قانون دوم – هرس بتا

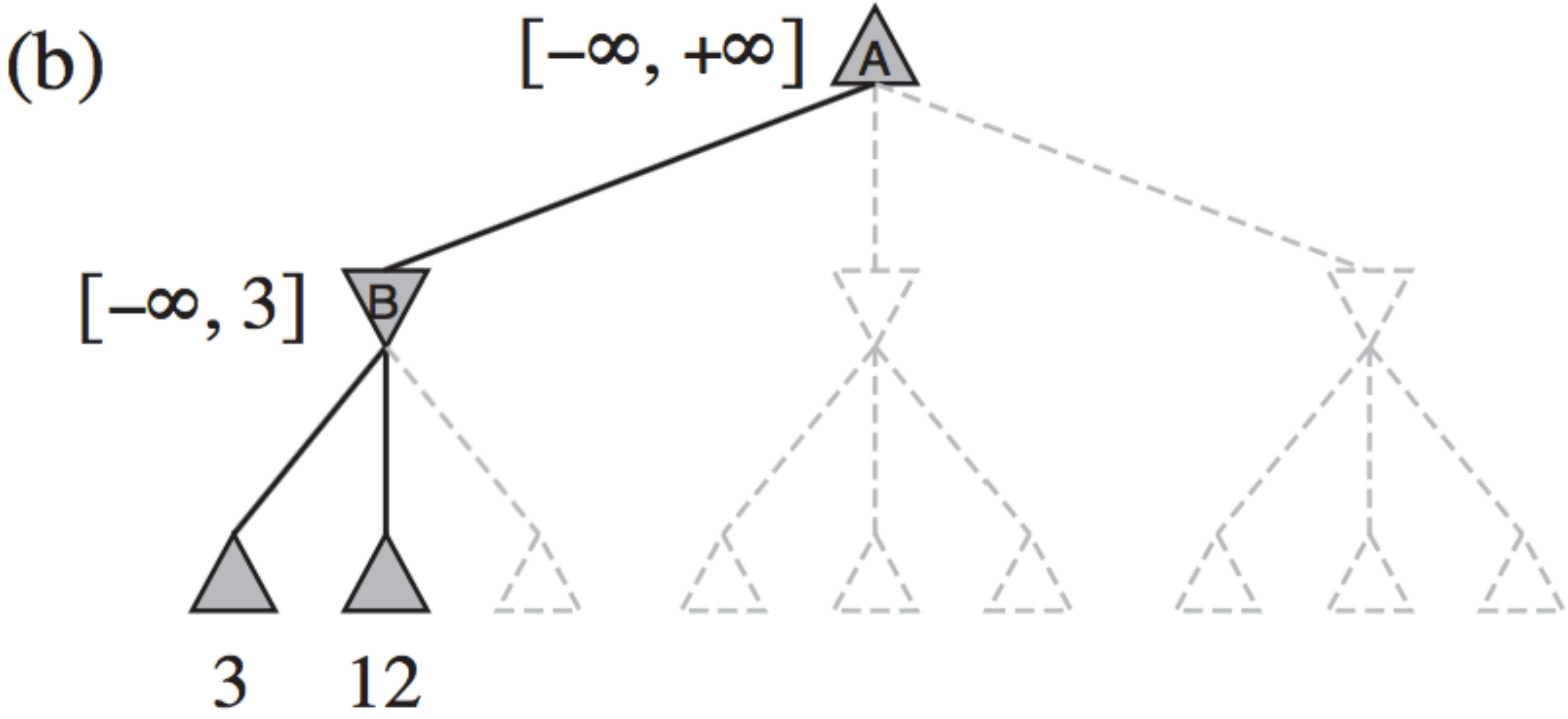


- عمل جستجو تحت گره MAX که مقدار آلفای آن بزرگ‌تر یا مساوی مقدار بتای هر گره MIN اجداد آن گره است، قطع می‌شود.
- در این حالت، مقداری که تا به‌حال به گره‌ی MAX نسبت داده شده به‌عنوان مقدار نهایی آلفای آن گره انتخاب خواهد شد.

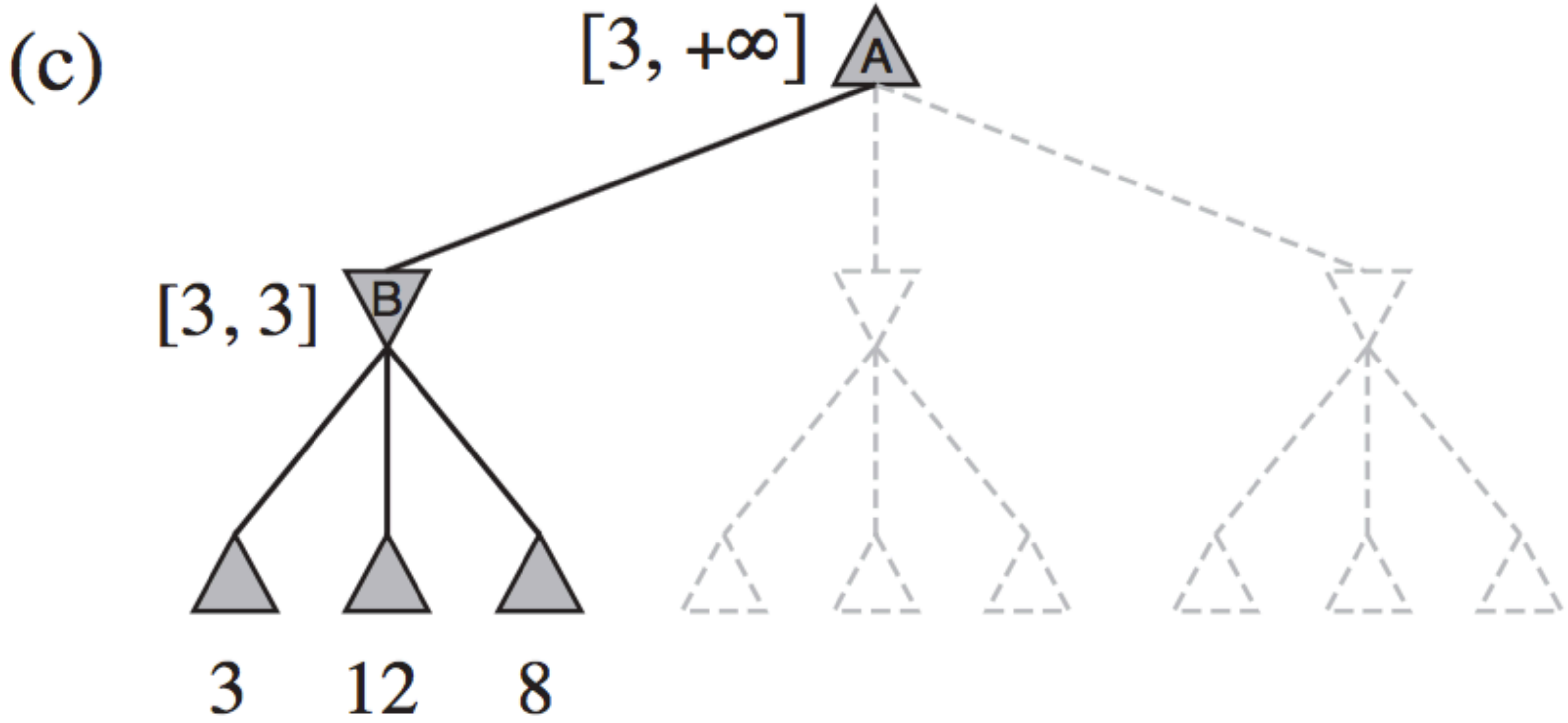
آلفا–بتا (مثال ۱)



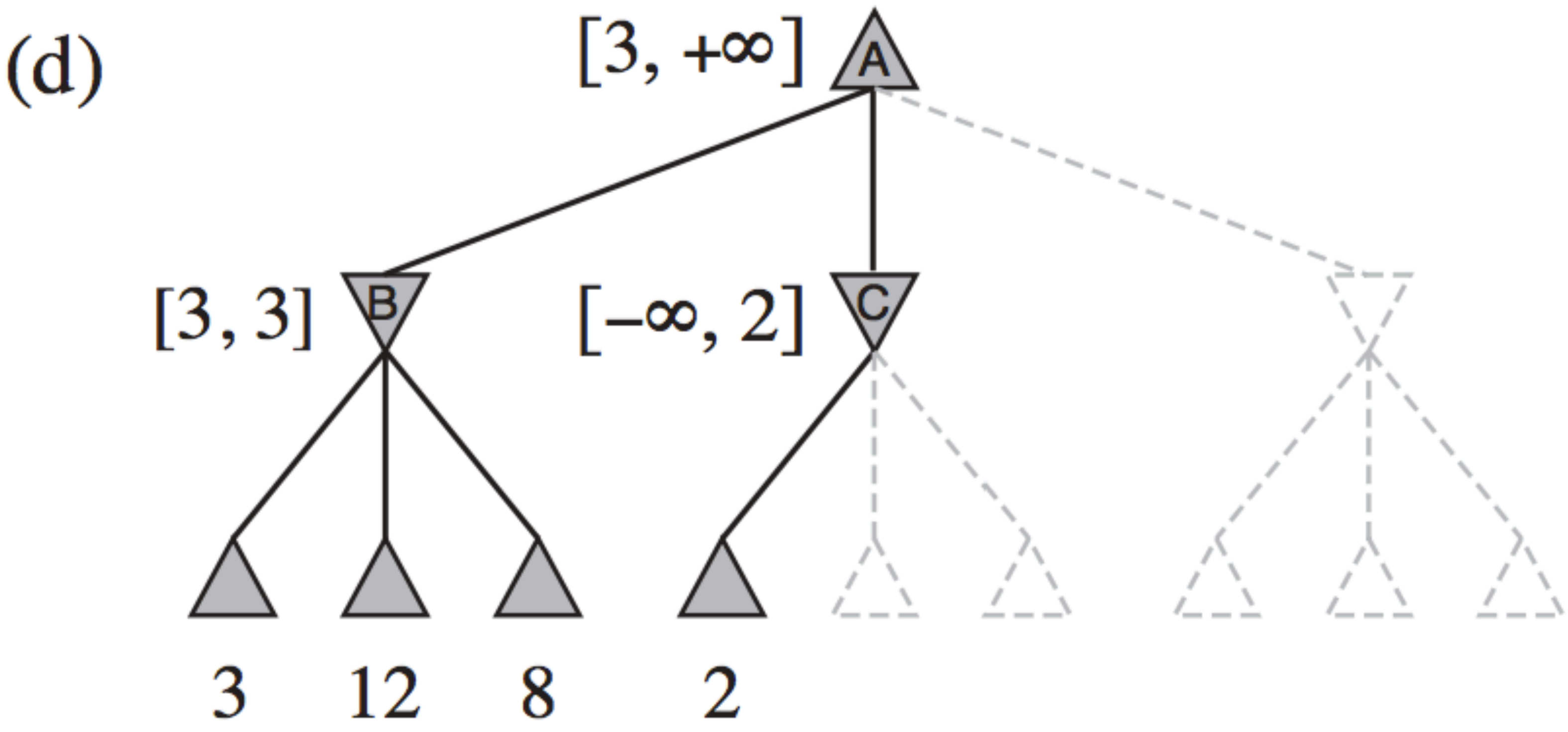
آلفا– بتا (مثال ۱)



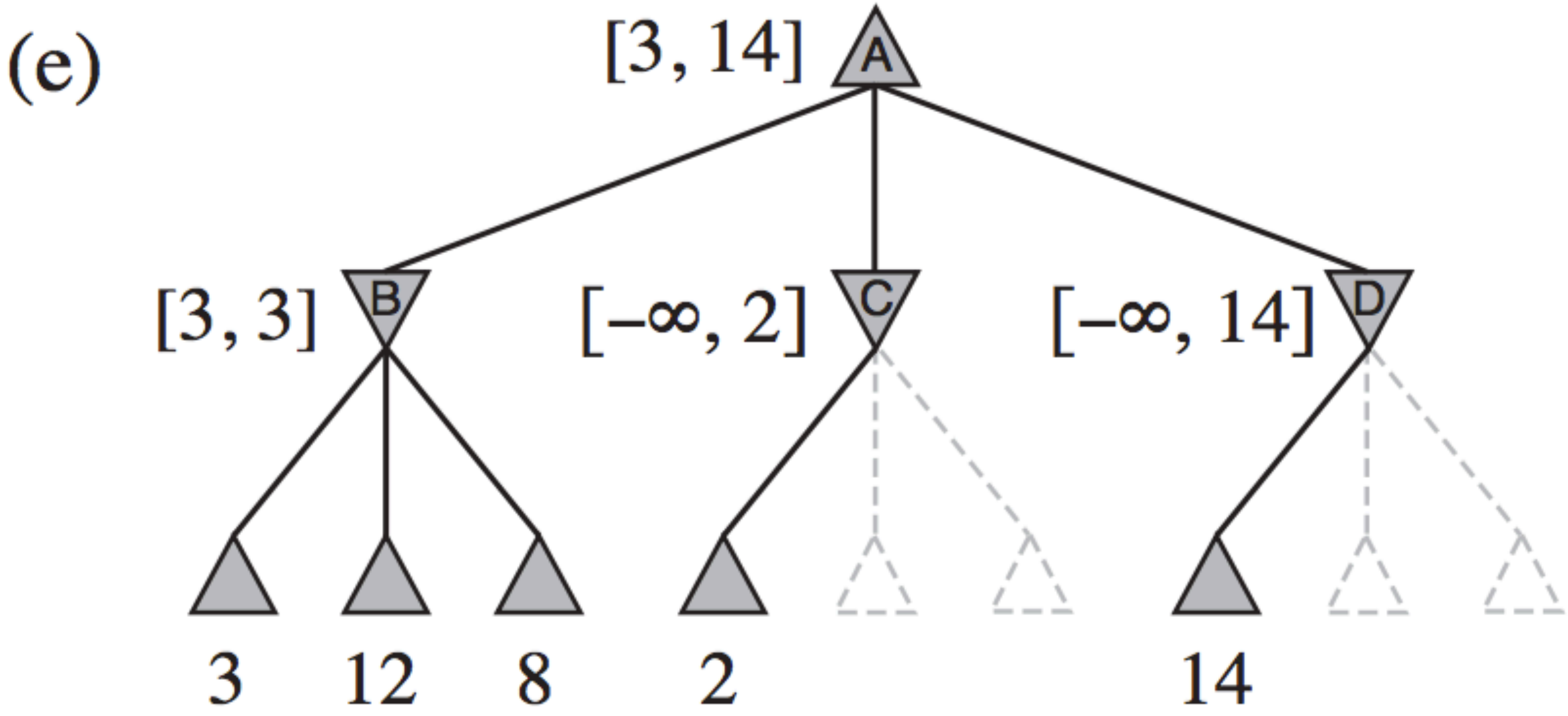
آلفا– بتا (مثال ۱)



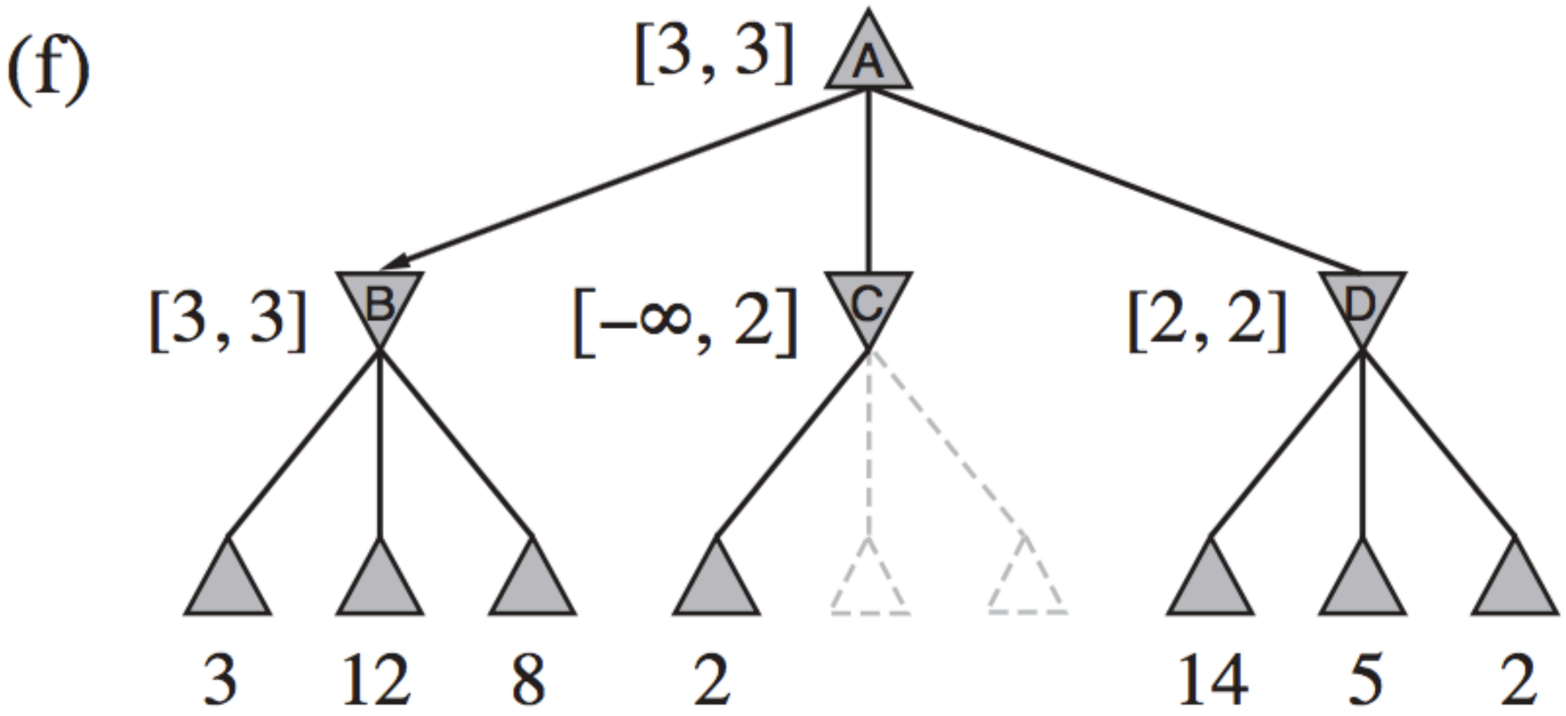
آلفا–بتا (مثال ۱)



آلفا– بتا (مثال ۱)



آلفا–بتا (مثال ۱)



الگوریتم آلفا–بتا

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

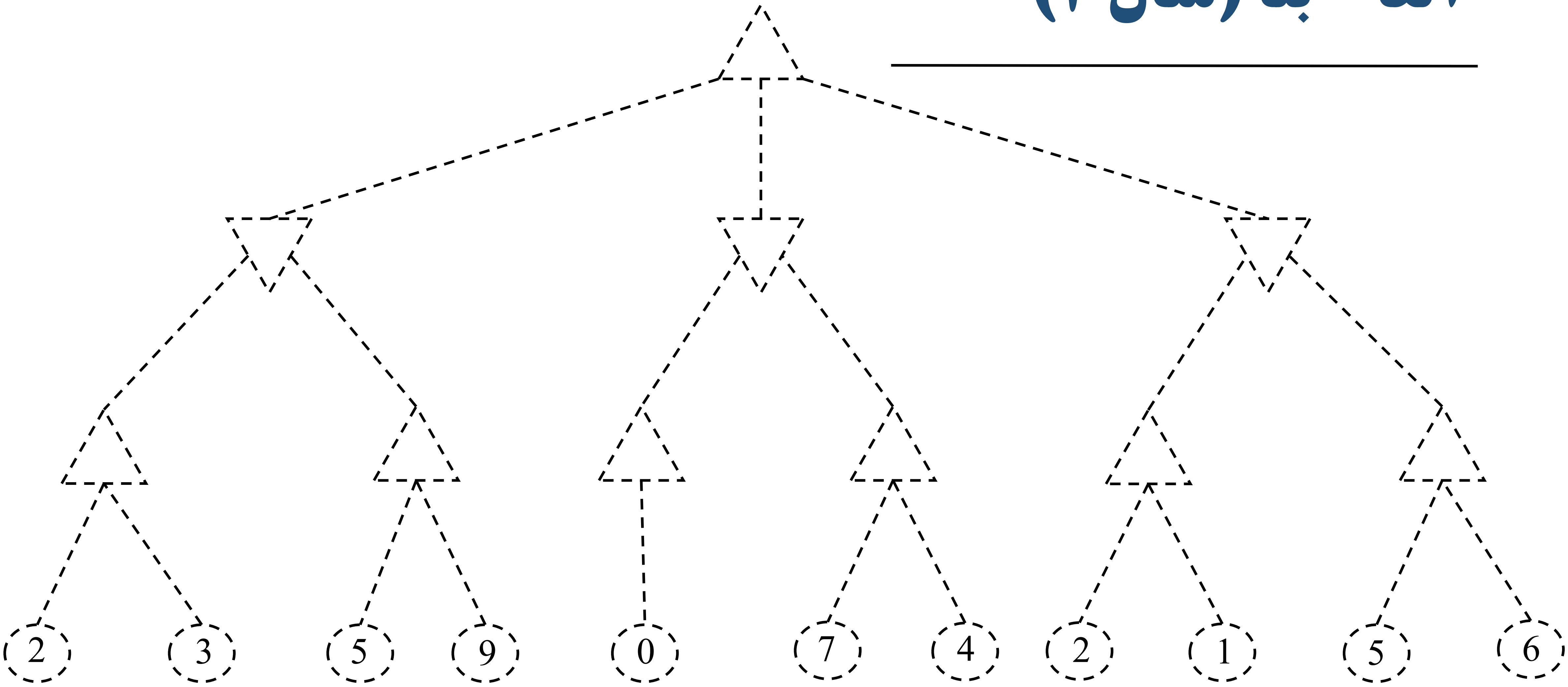
```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

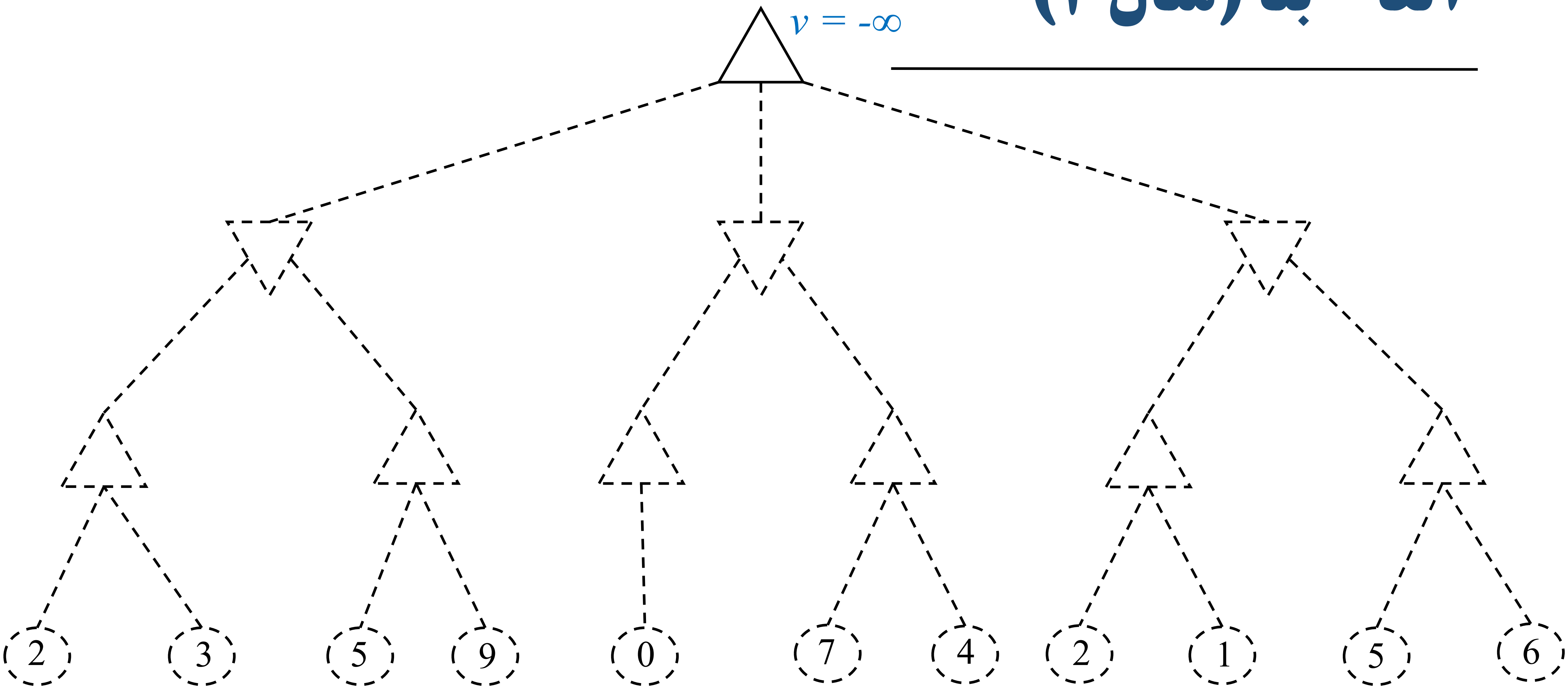
استفاده از الگوریتم آلفا–بتا

- آلفا: سودمندی بهترین (بیشترین) انتخابی که تا کنون در گره‌های مسیر مربوط به MAX پیدا شده است.
- بتا: سودمندی بهترین (کم‌ترین) انتخابی که تا کنون در گره‌های مسیر مربوط به MIN پیدا شده است
- به‌روزرسانی آلفا و بتا در طول فرایند جستجو و هرس کردن براساس آن‌ها
  - هرس بتا: برای یک نود MAX زمانی‌که مقدار یافت‌شده برای آن v بیشتر از مقدار بتای فعلی بود ($v \geq \beta$)، زیرشاخه‌هایش هرس می‌شوند.
  - هرس آلفا: برای یک نود MIN زمانی‌که مقدار یافت‌شده برای آن v کمتر از مقدار آلفای فعلی بود ($v \leq \alpha$)، زیرشاخه‌هایش هرس می‌شوند.

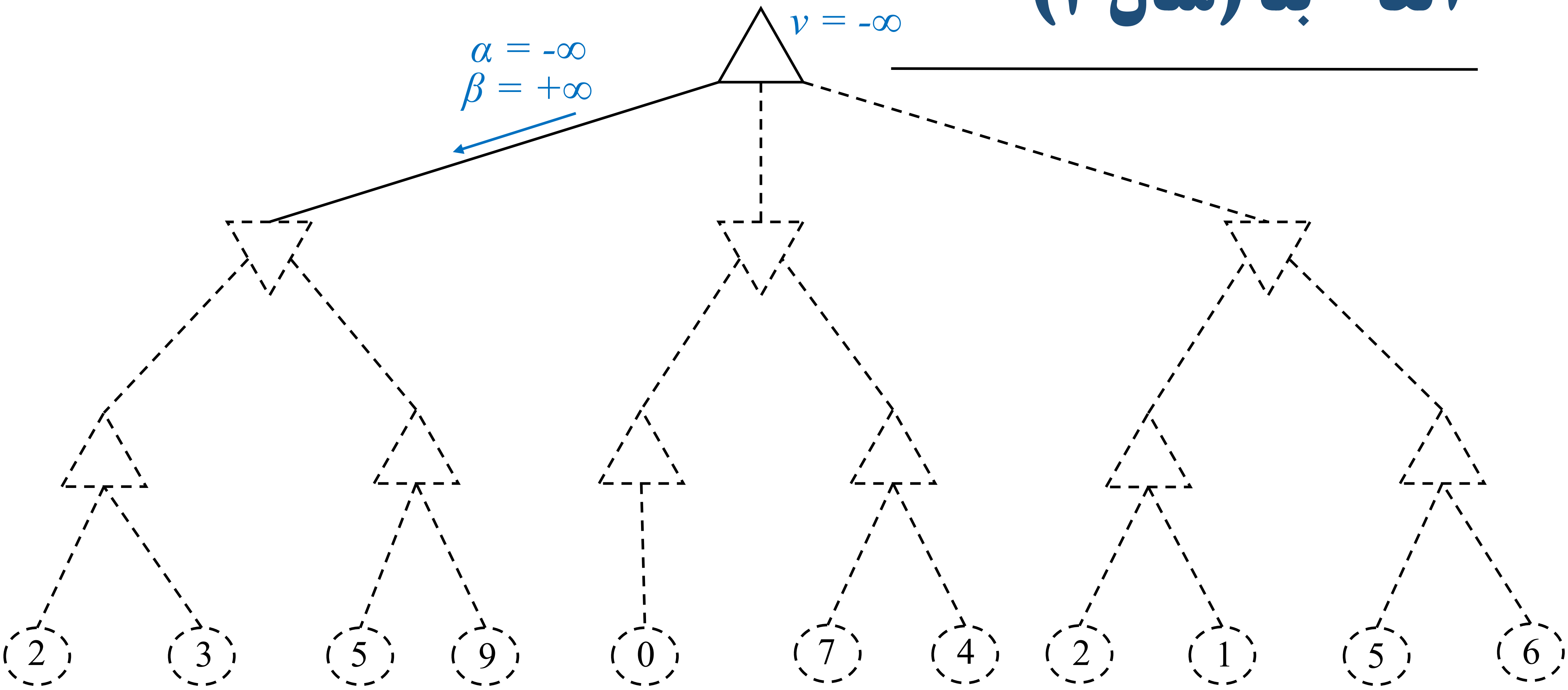
آلفا– بتا (مثال ۲)



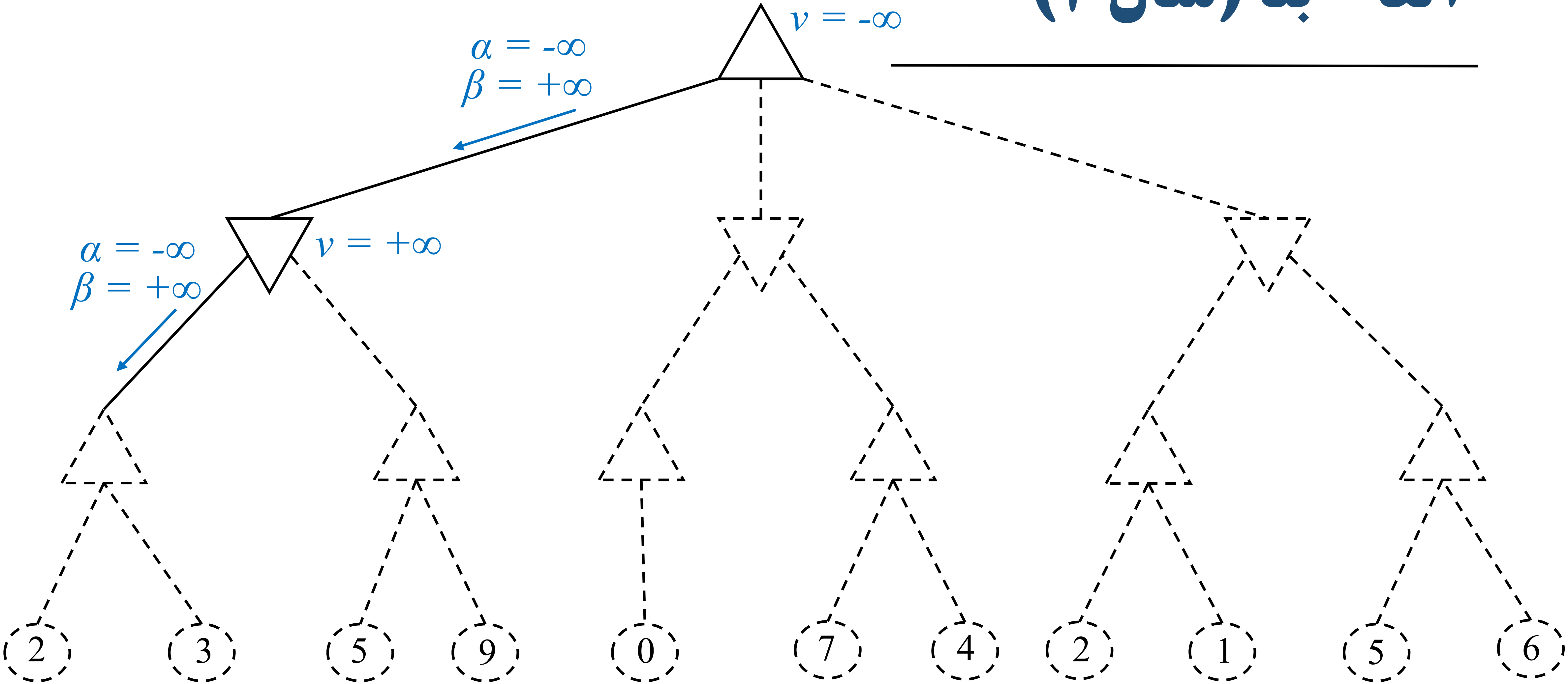
آلفا–بتا (مثال ۲)



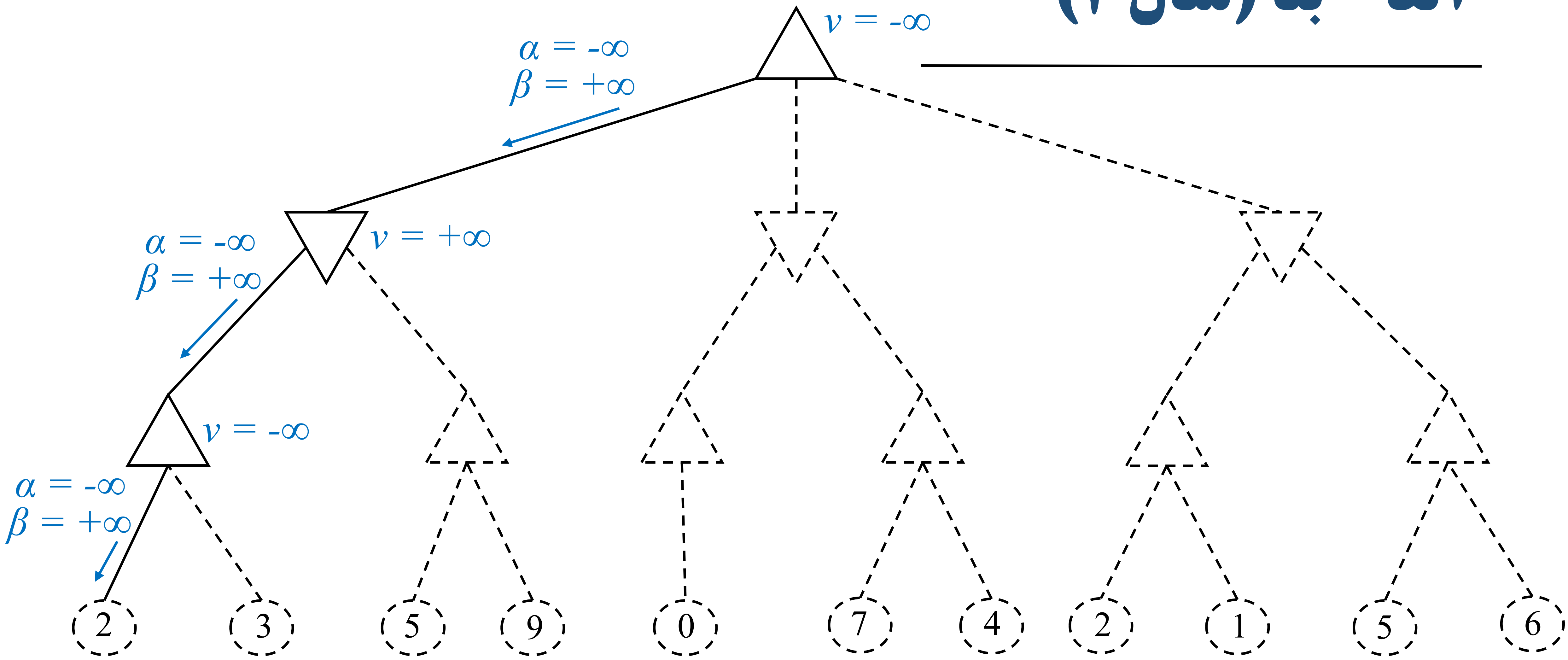
آلفا–بتا (مثال ۲)



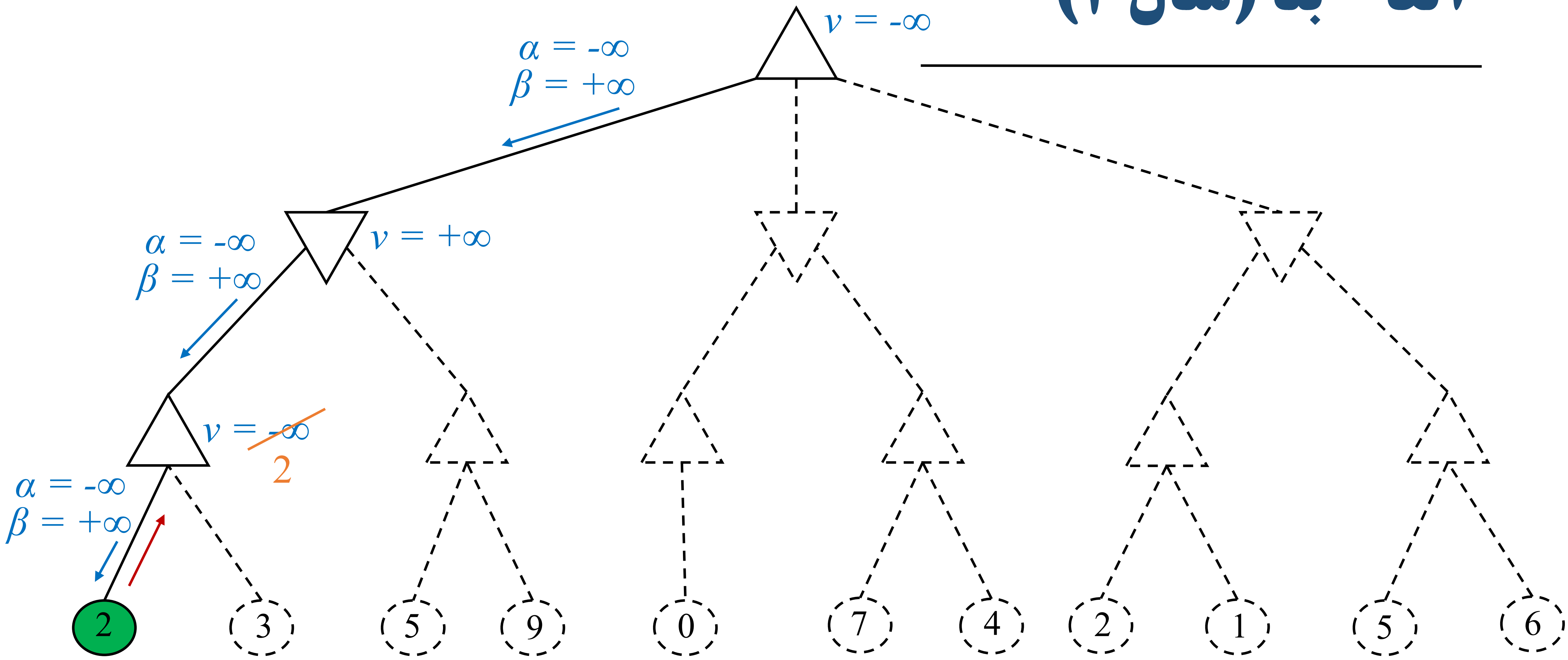
آلفا—بتا (مثال ۲)



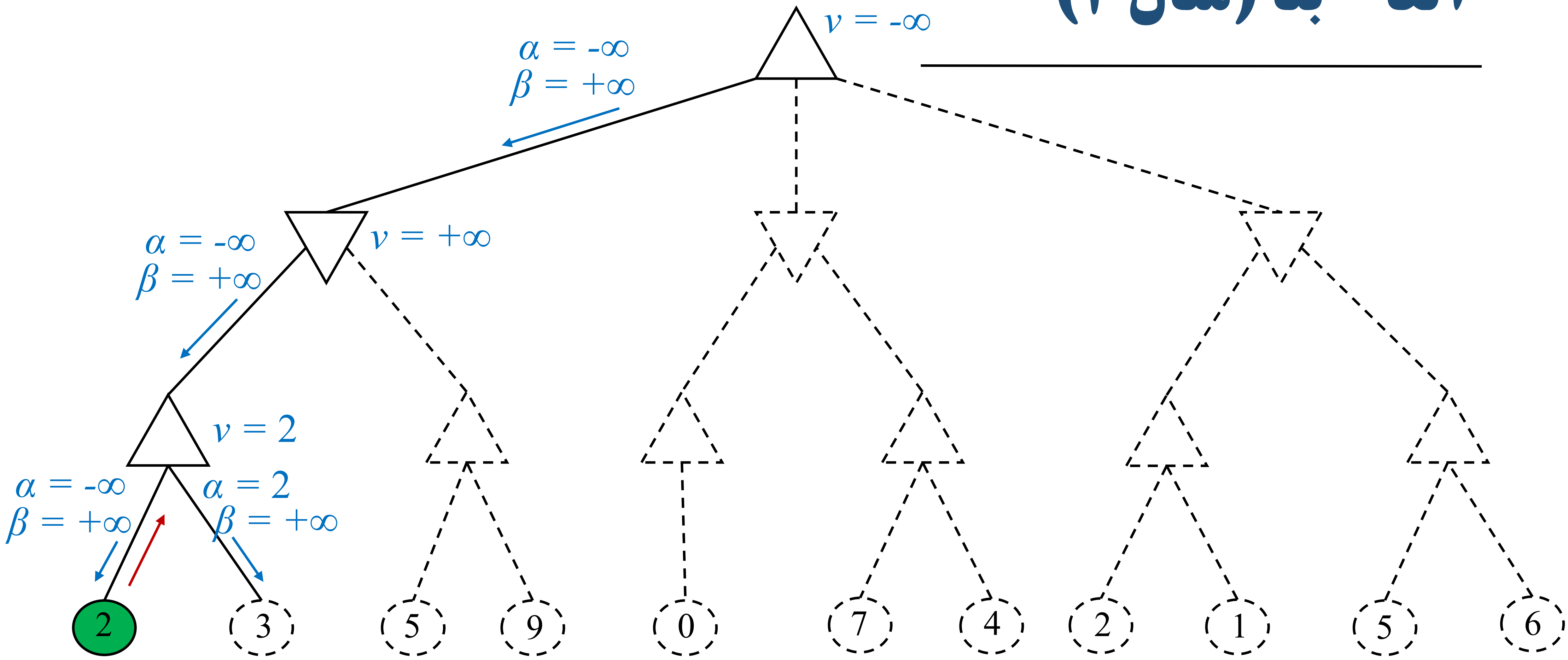
آلفا–بتا (مثال ۲)



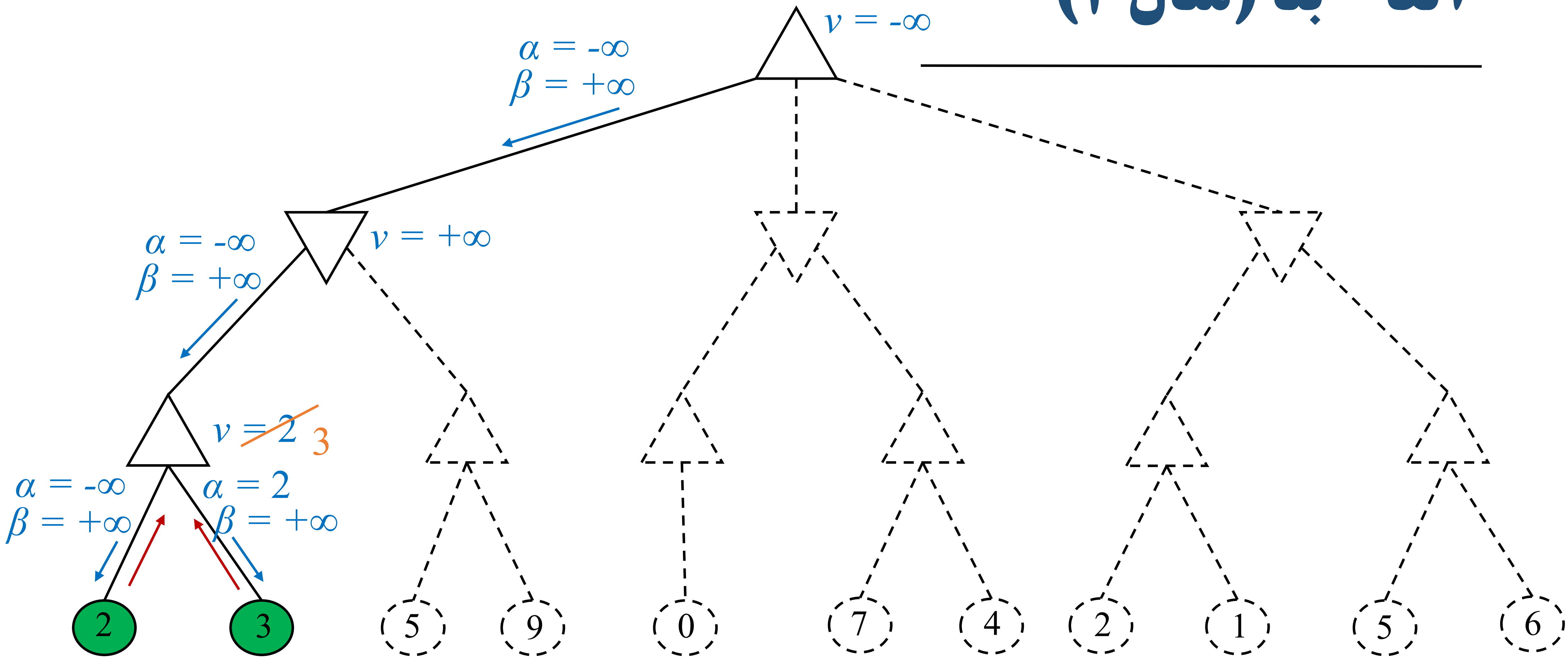
آلفا–بتا (مثال ۲)



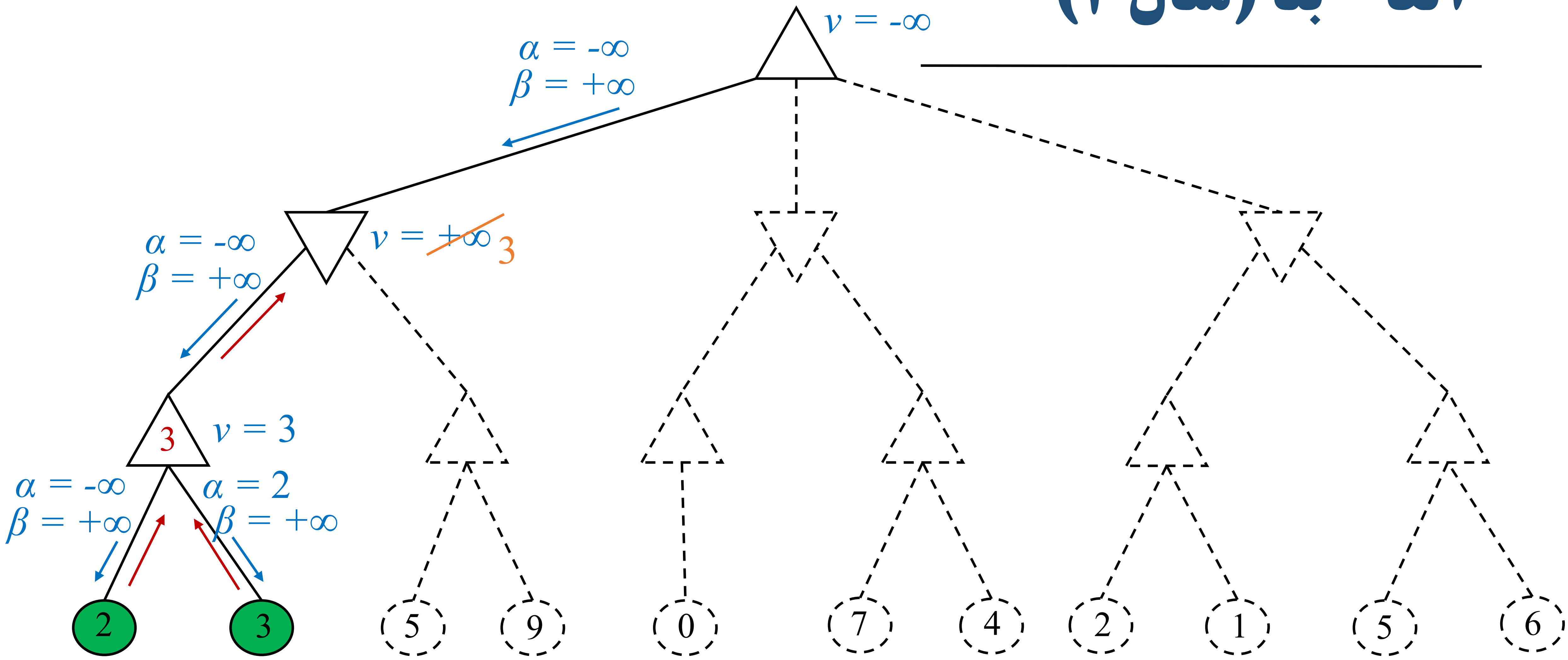
آلفا– بتا (مثال ۲)



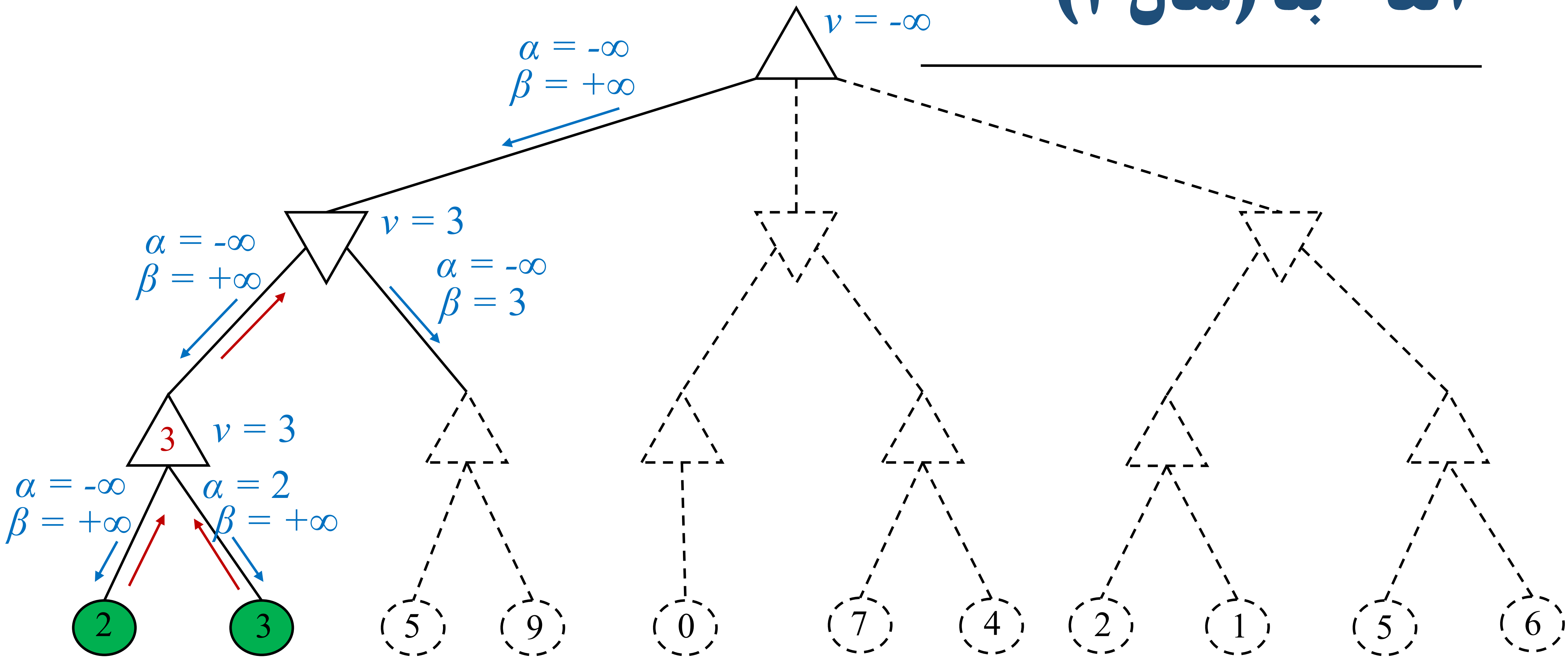
آلفا – بتا (مثال ۲)



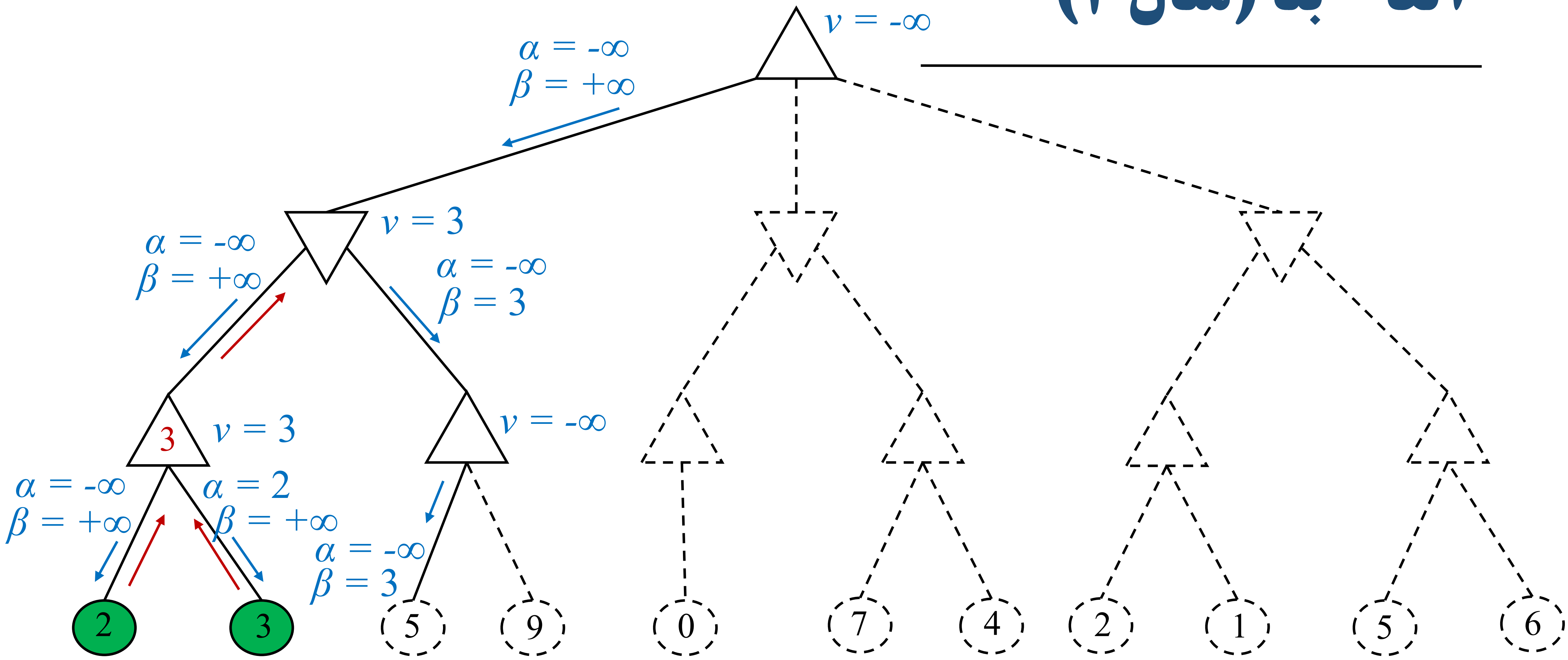
آلفا–بتا (مثال ۲)



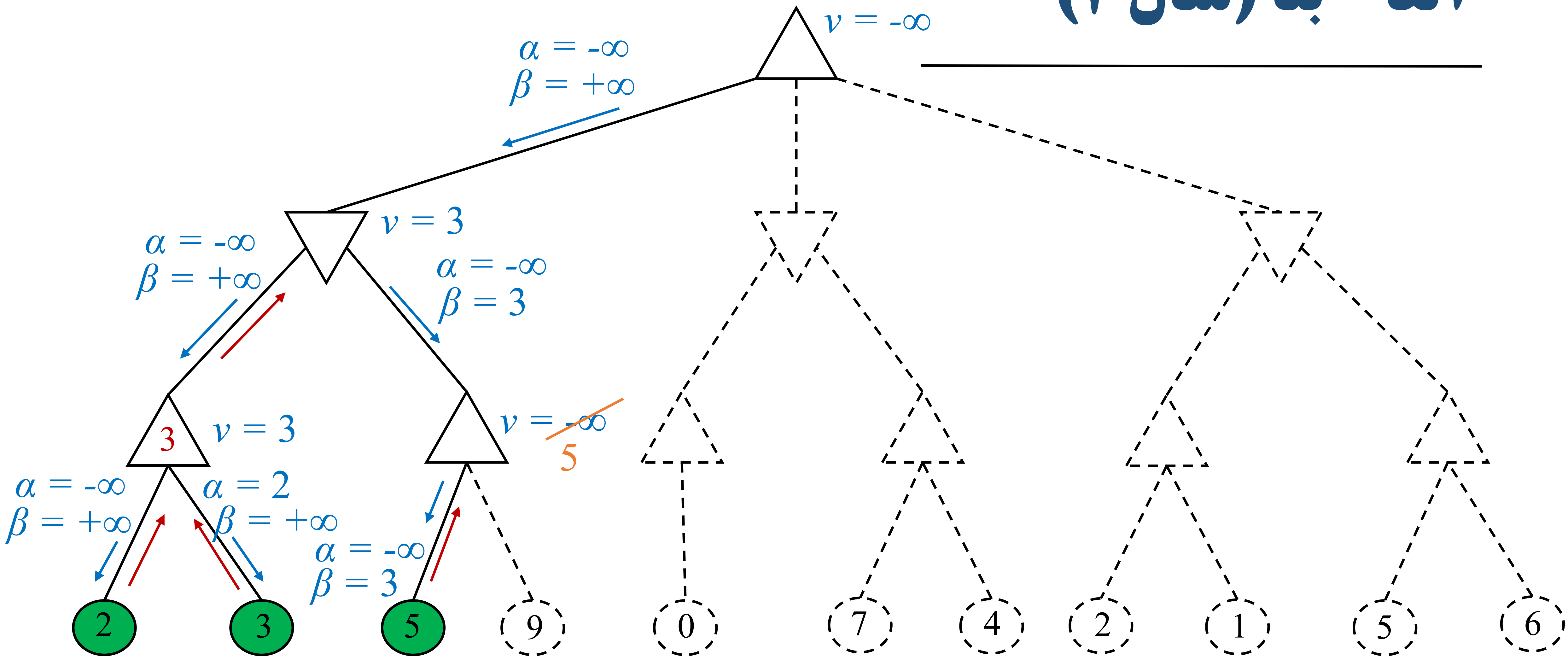
آلفا–بتا (مثال ۲)



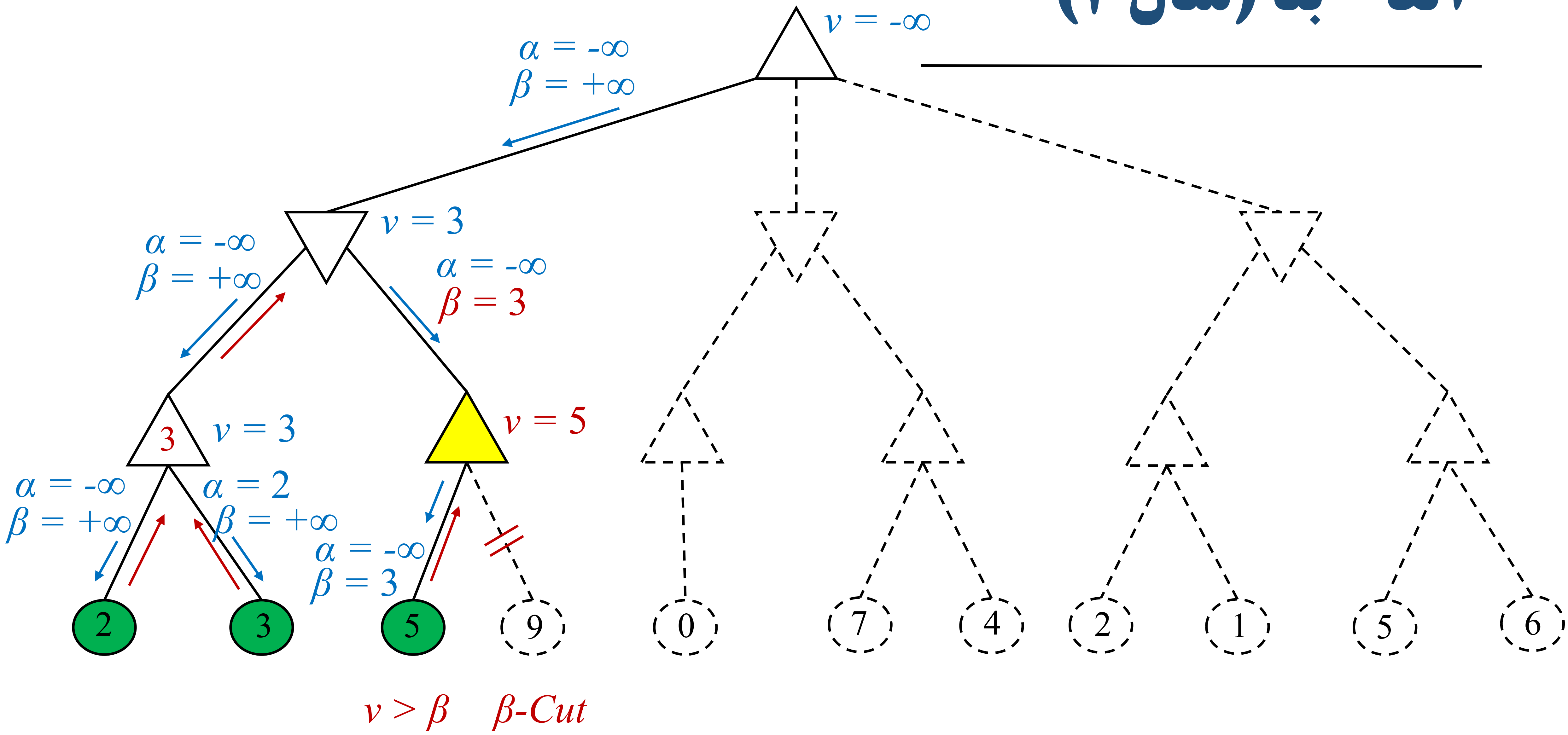
آلفا– بتا (مثال ۲)



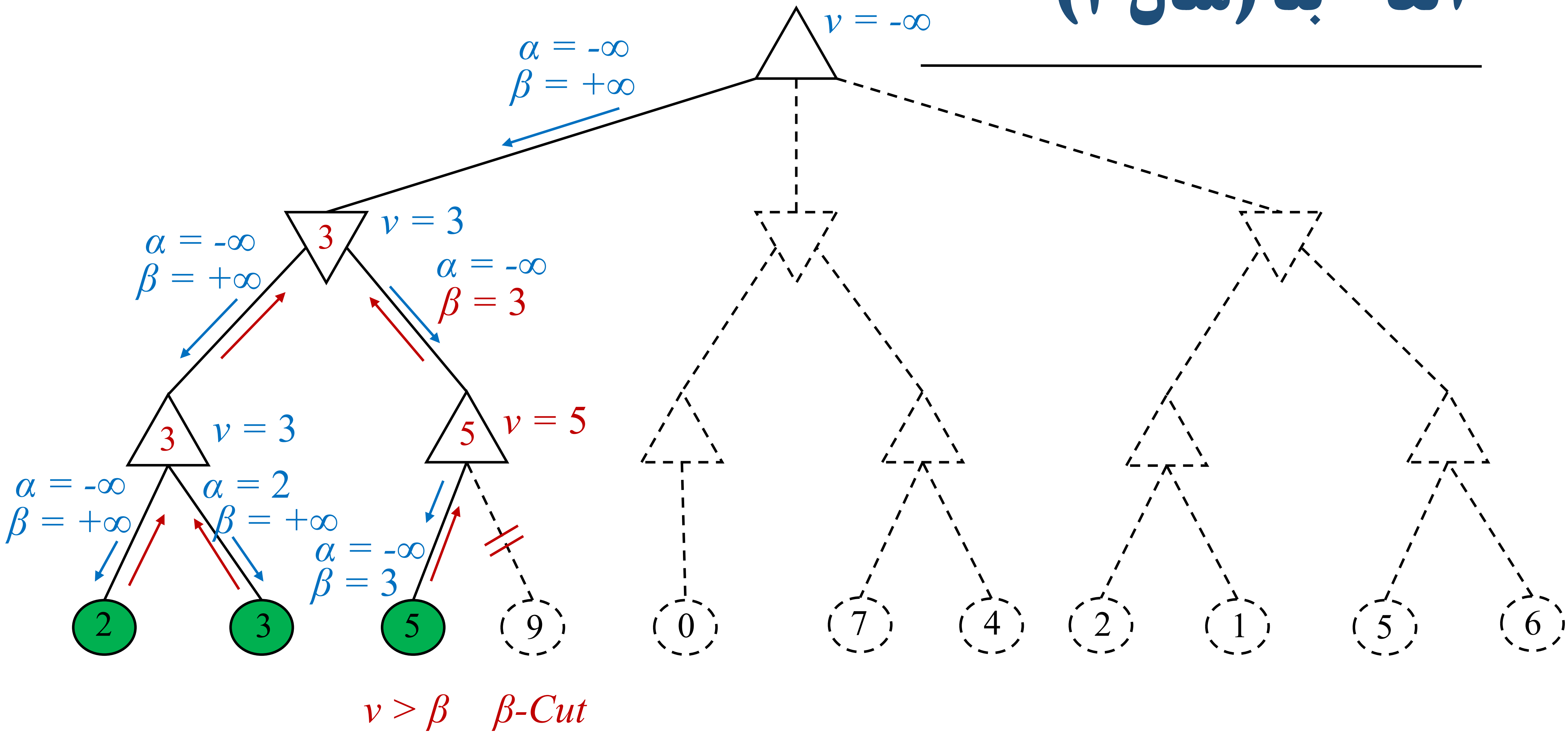
آلفا– بتا (مثال ۲)



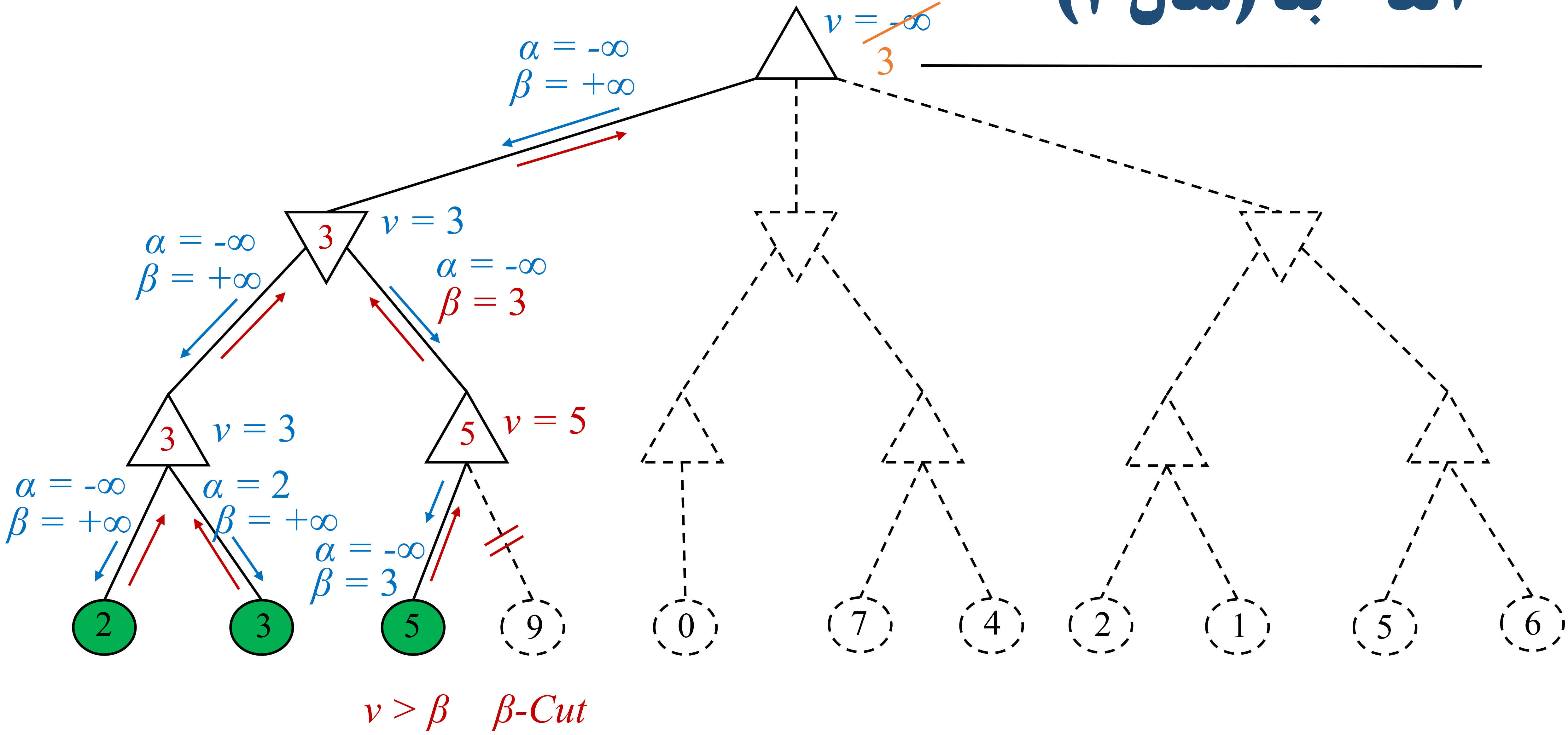
آلفا– بتا (مثال ۲)



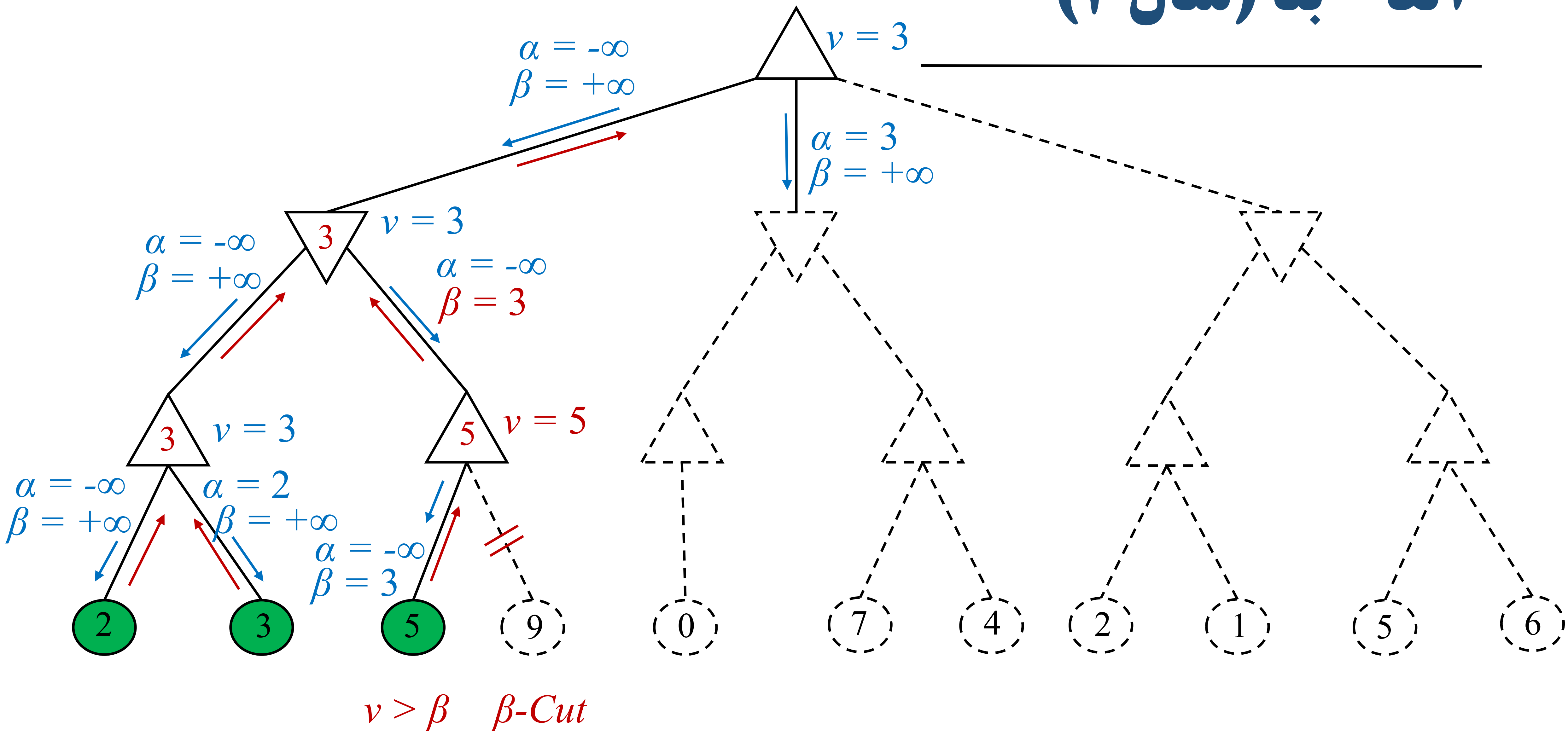
آلفا–بتا (مثال ۲)



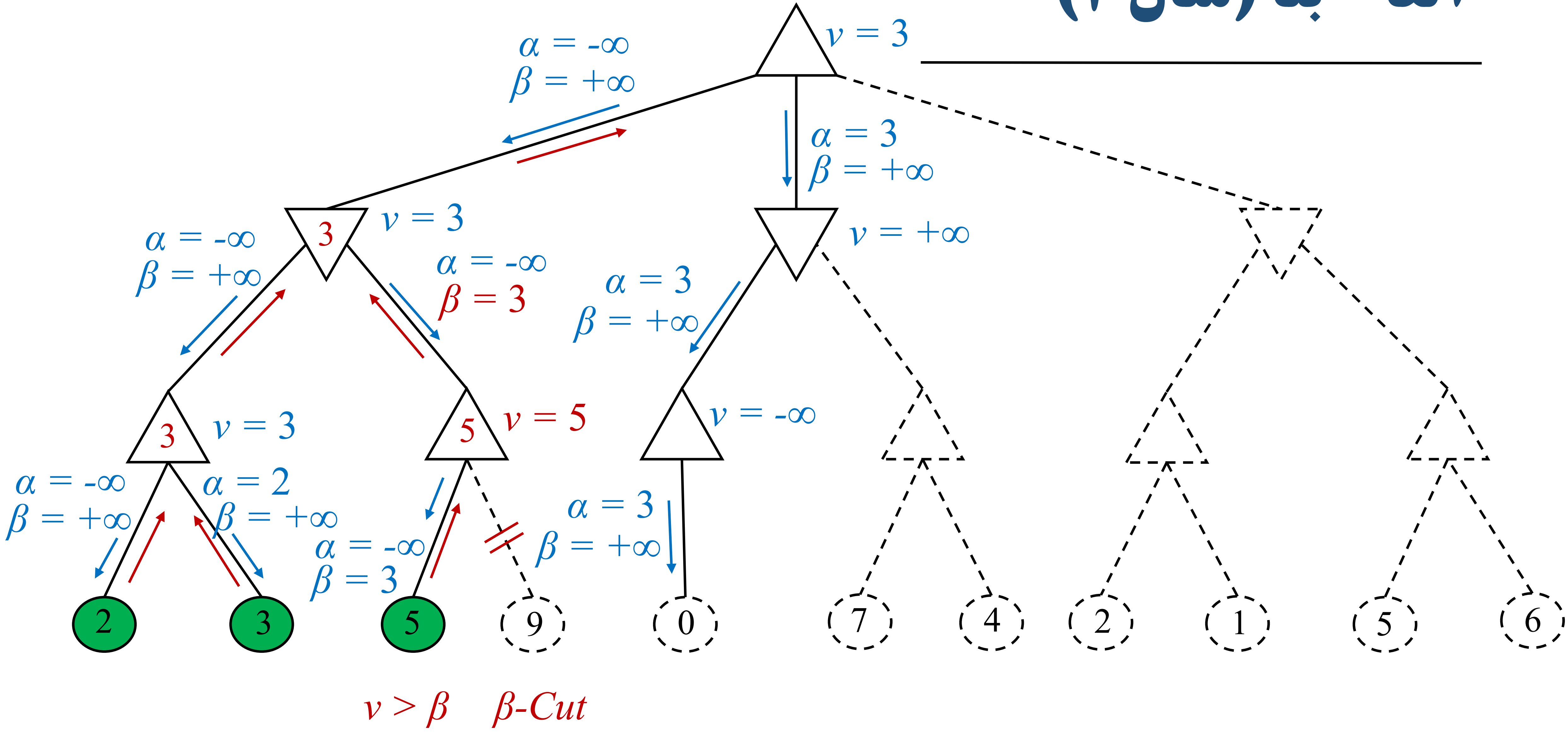
آلفا– بتا (مثال ۲)



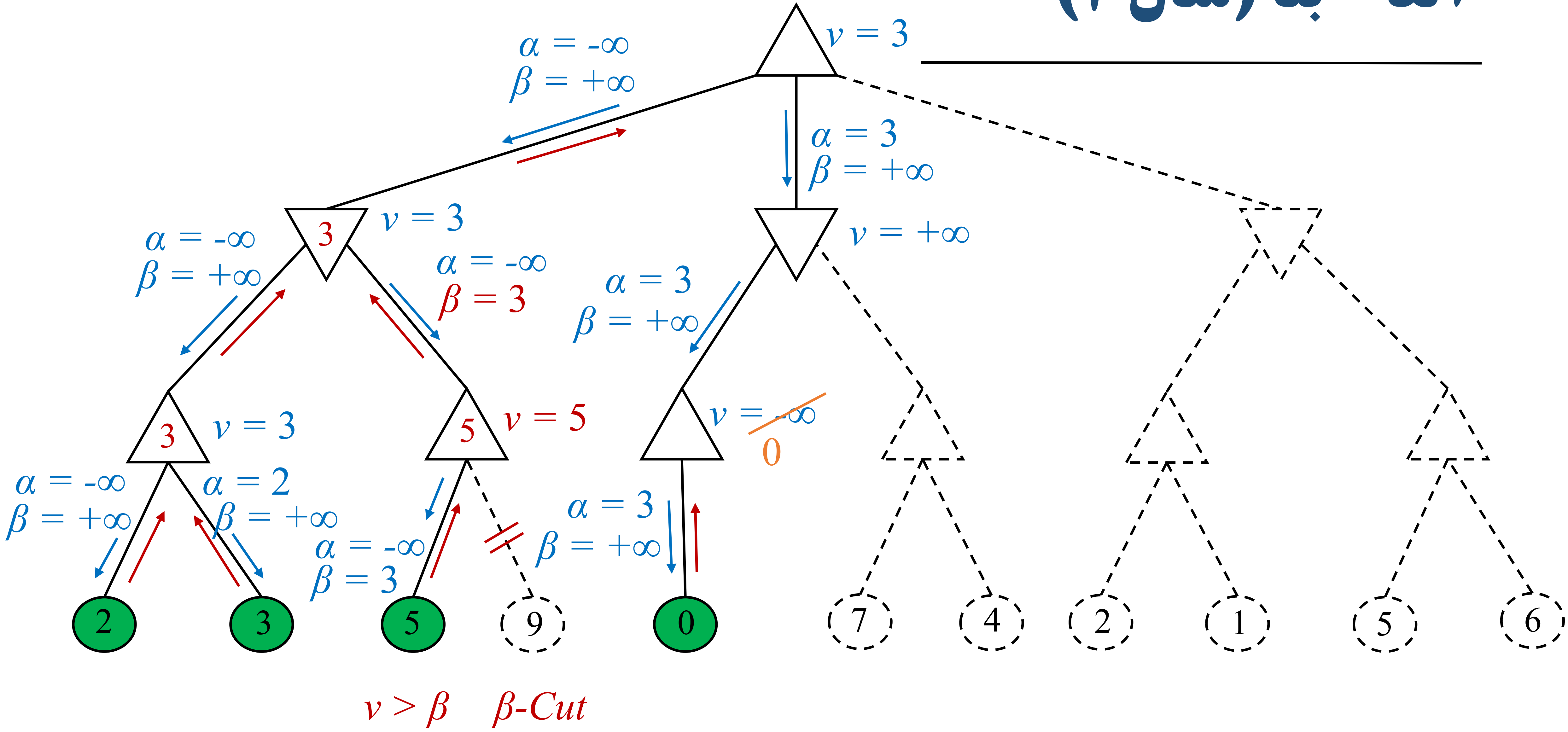
آلفا– بتا (مثال ۲)



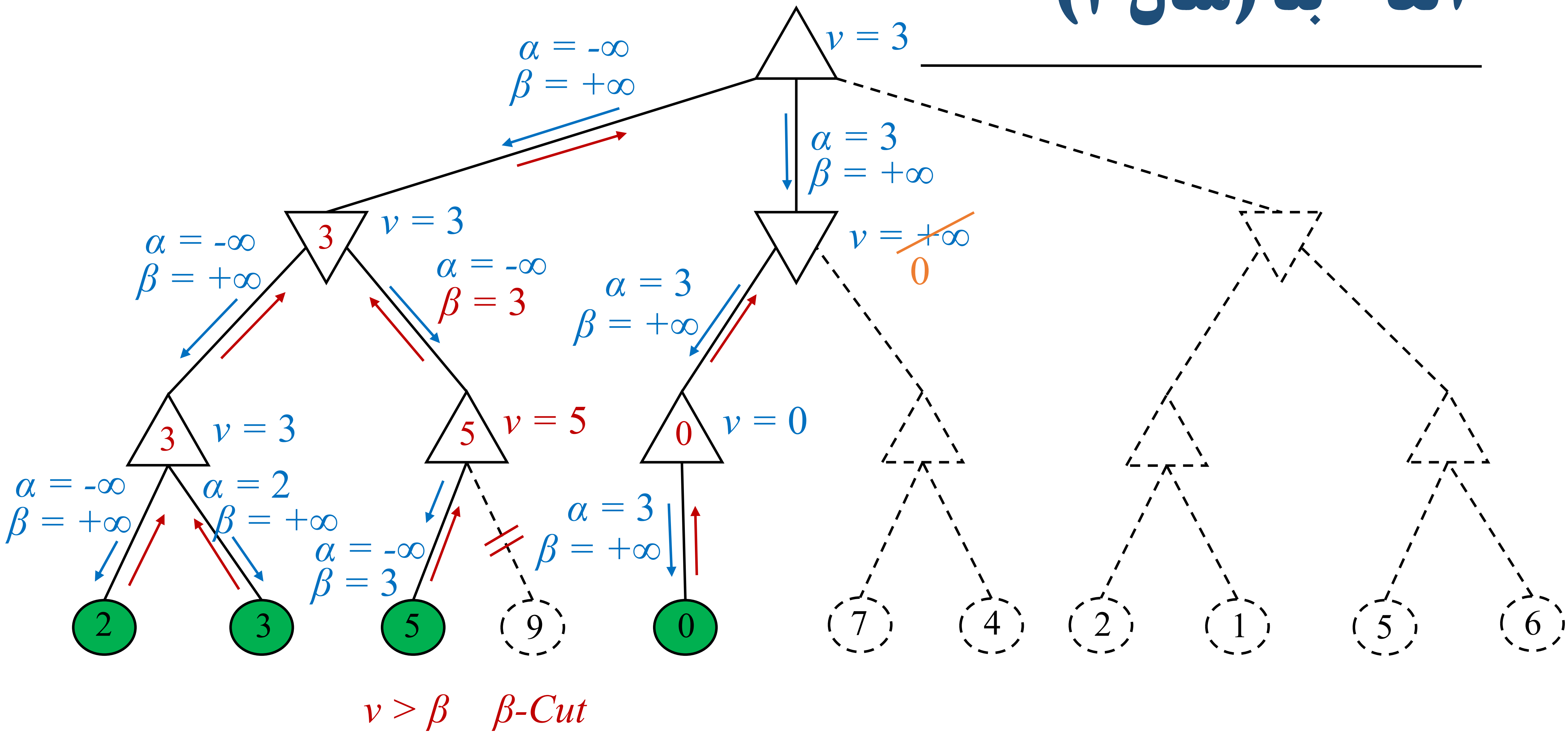
آلفا– بتا (مثال ۲)



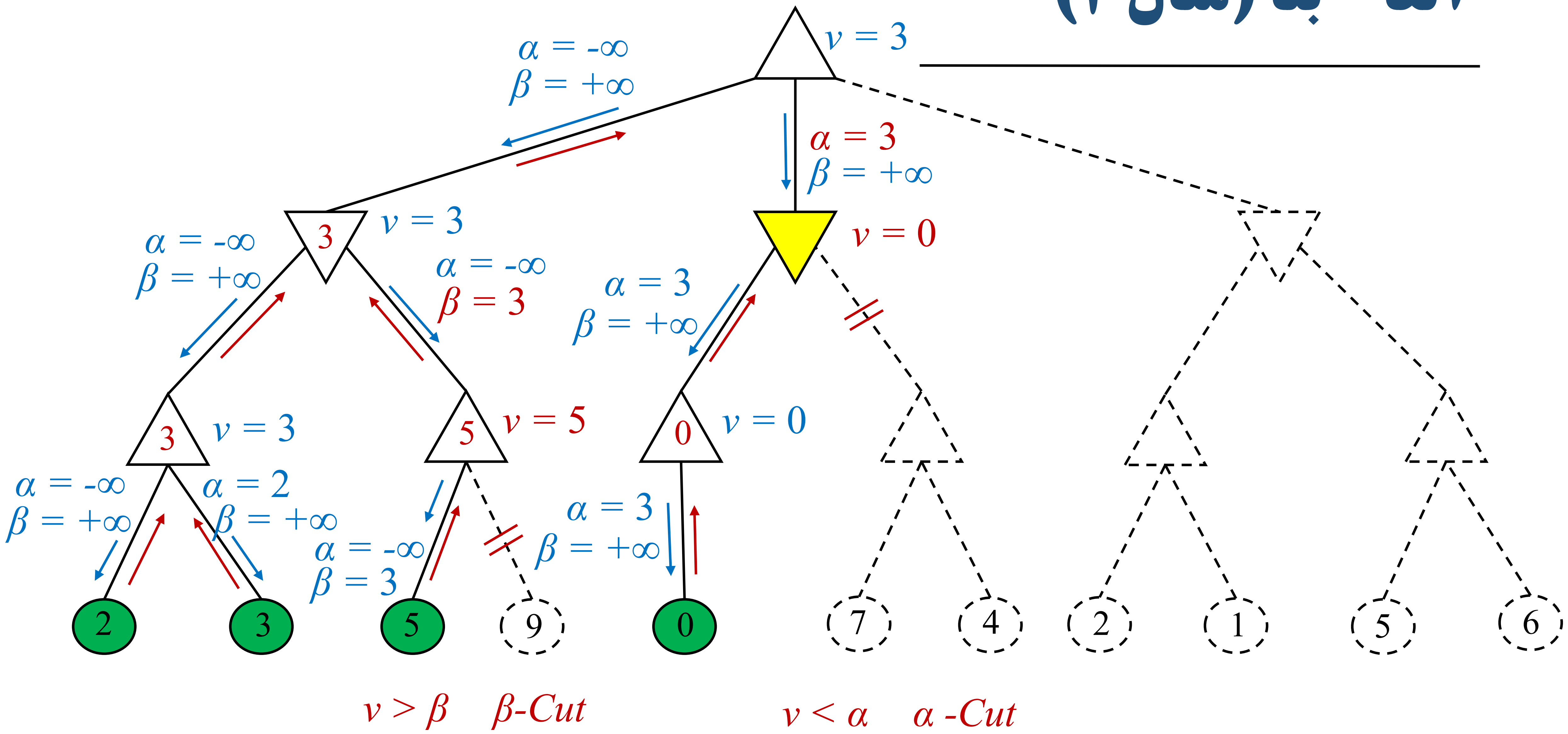
آلفا– بتا (مثال ۲)



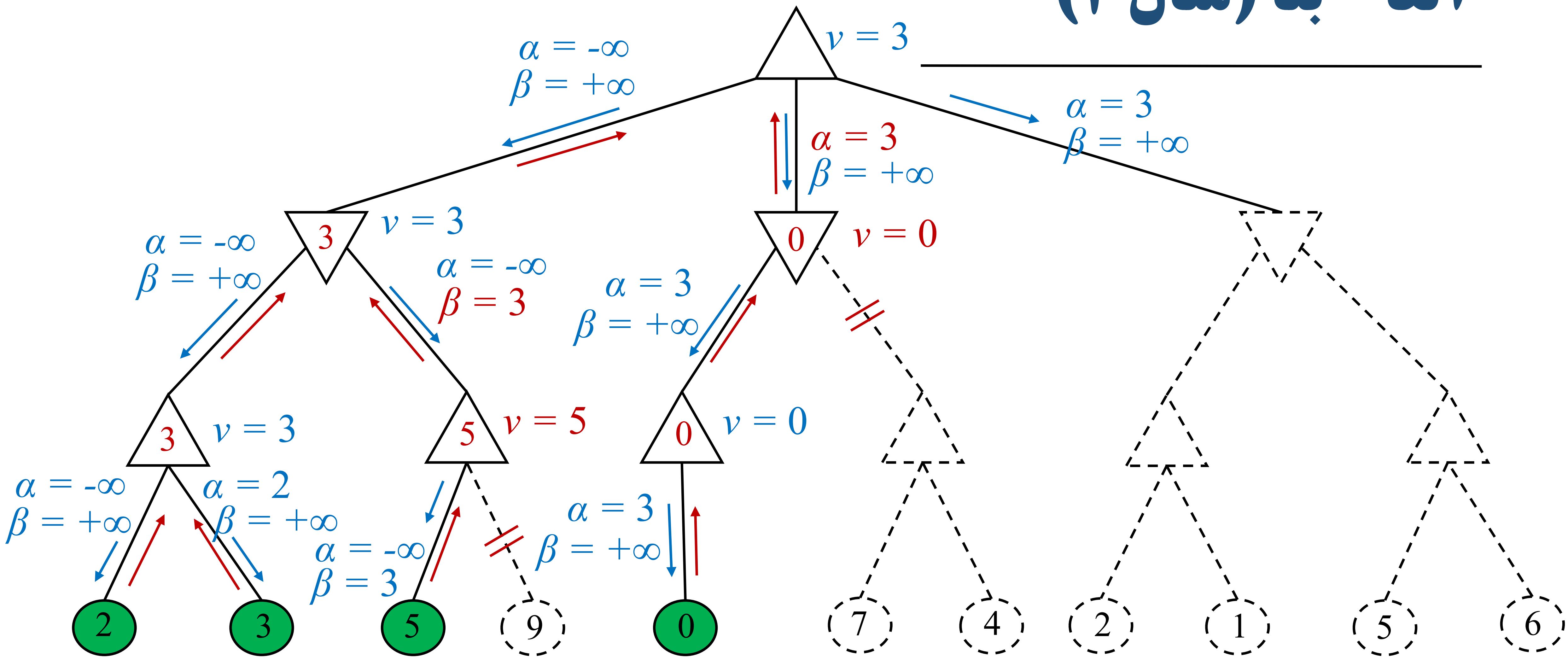
آلفا– بتا (مثال ۲)



آلفا– بتا (مثال ۲)



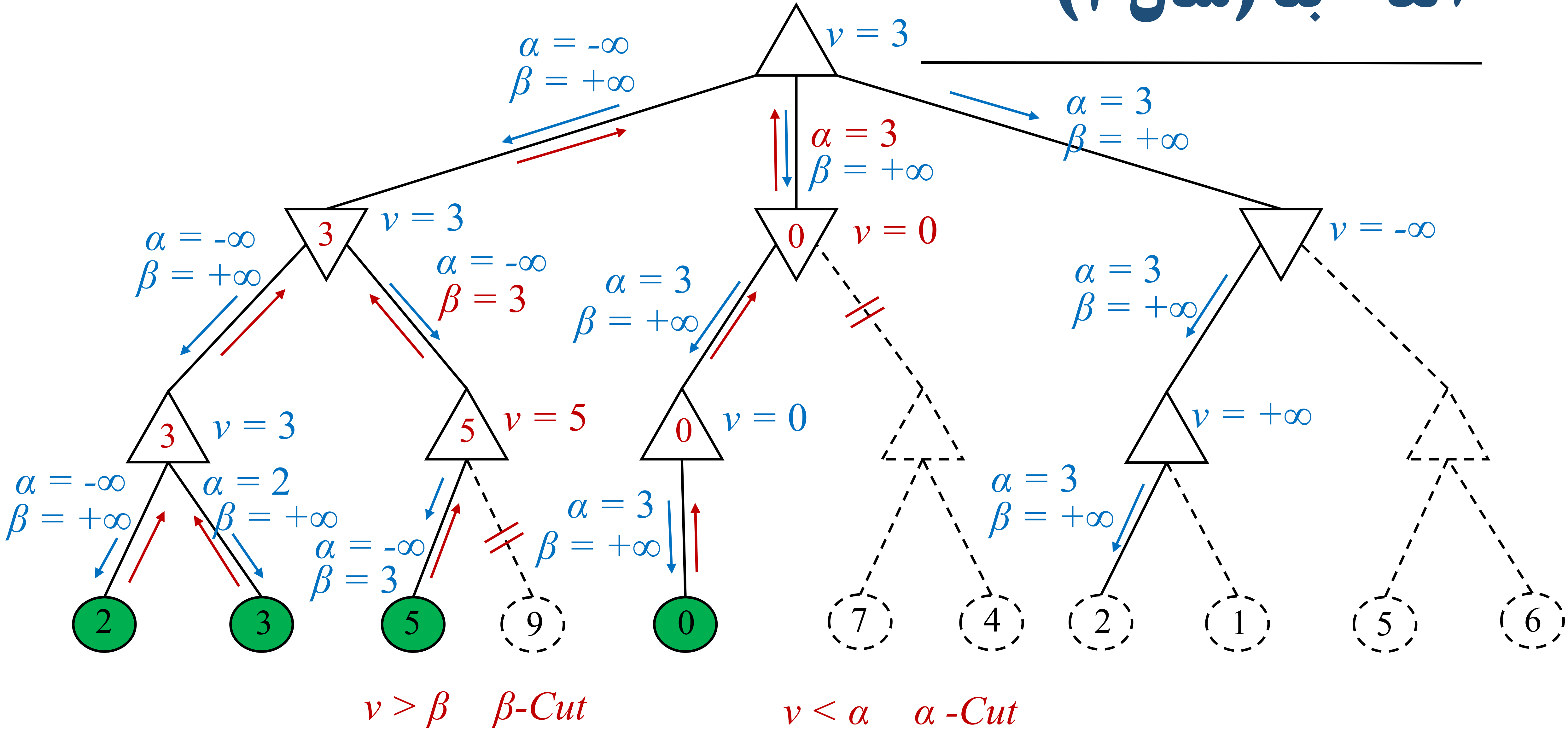
آلفا– بتا (مثال ۲)



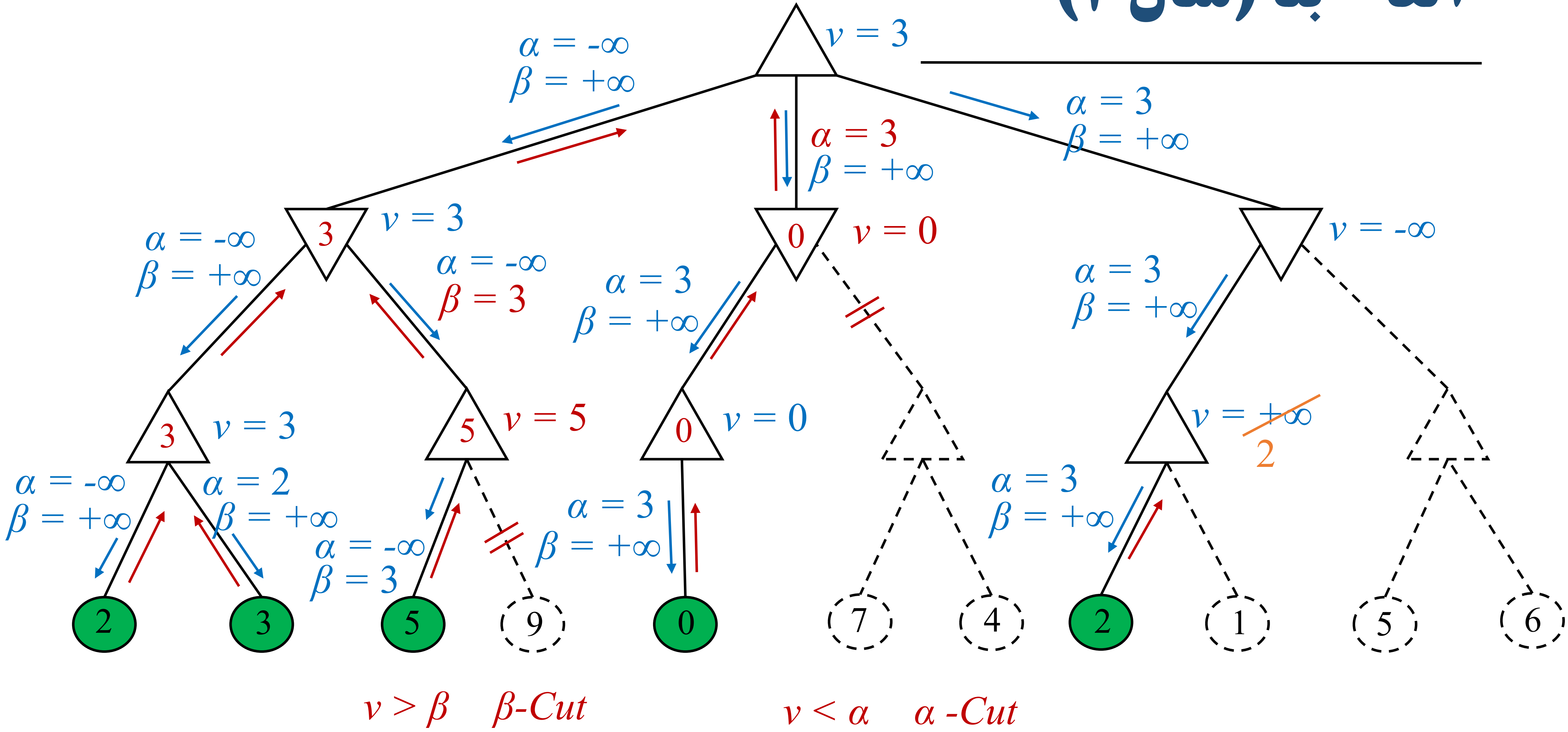
$v > \beta$ β-Cut

$v < \alpha$ α -Cut

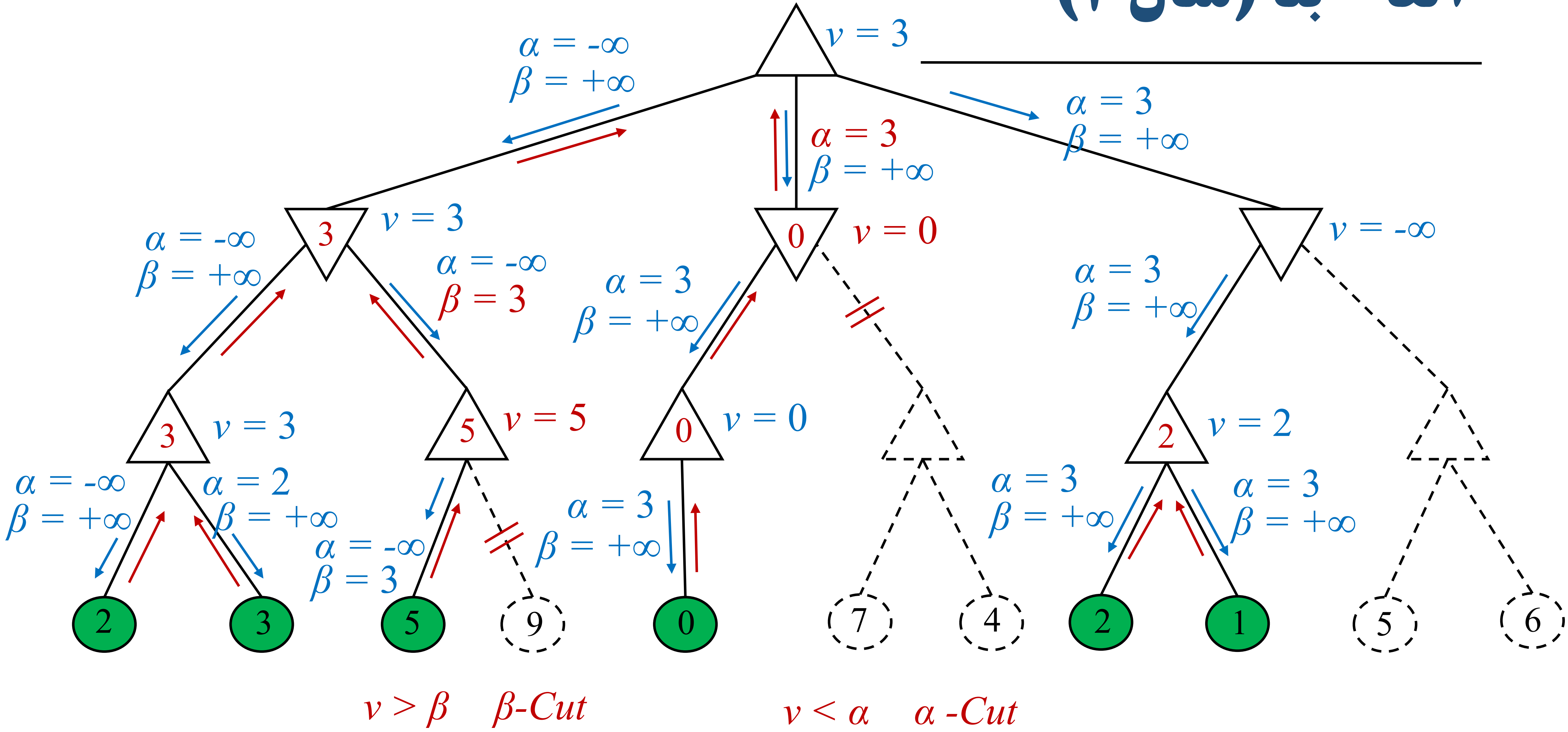
آلفا–بتا (مثال ۲)



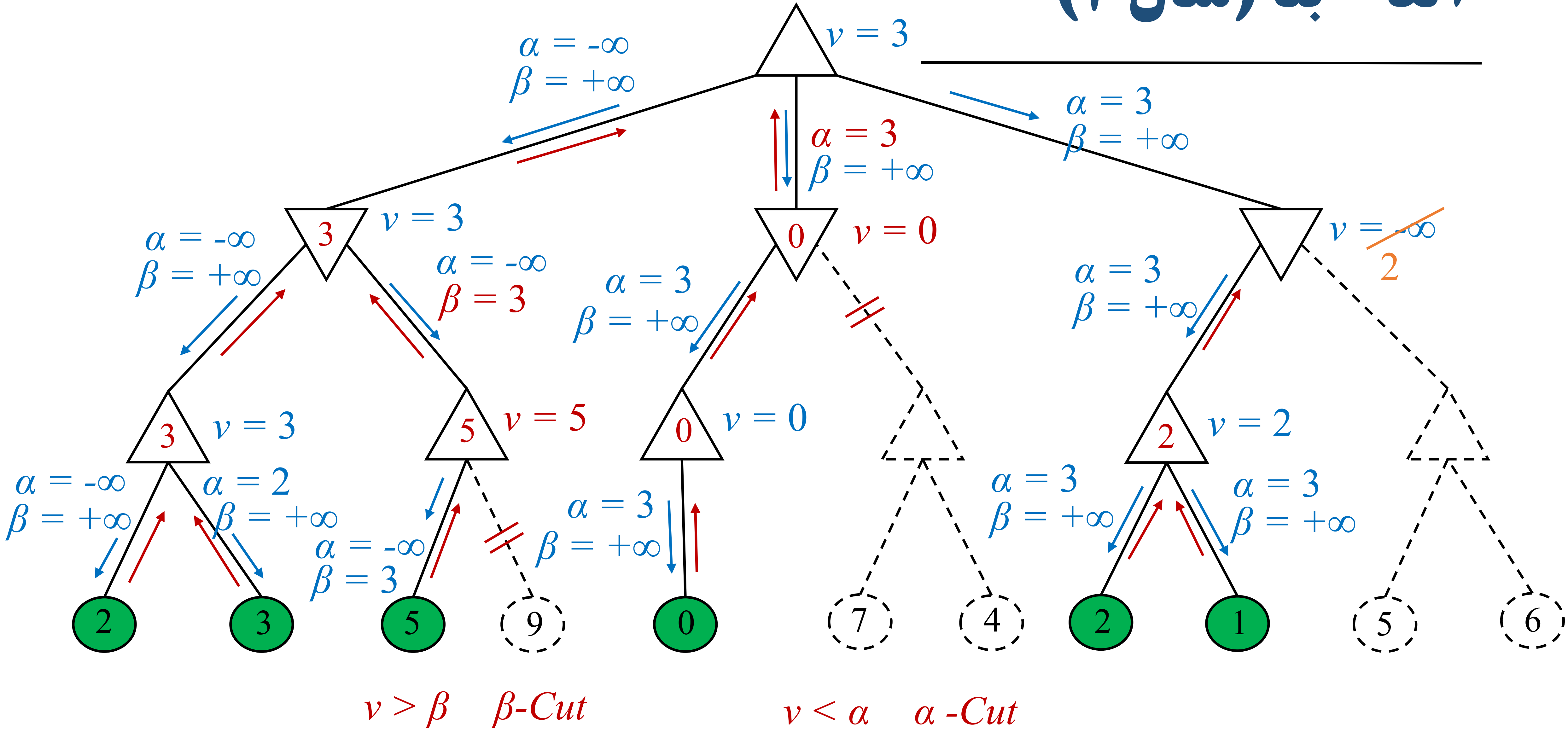
آلفا– بتا (مثال ۲)



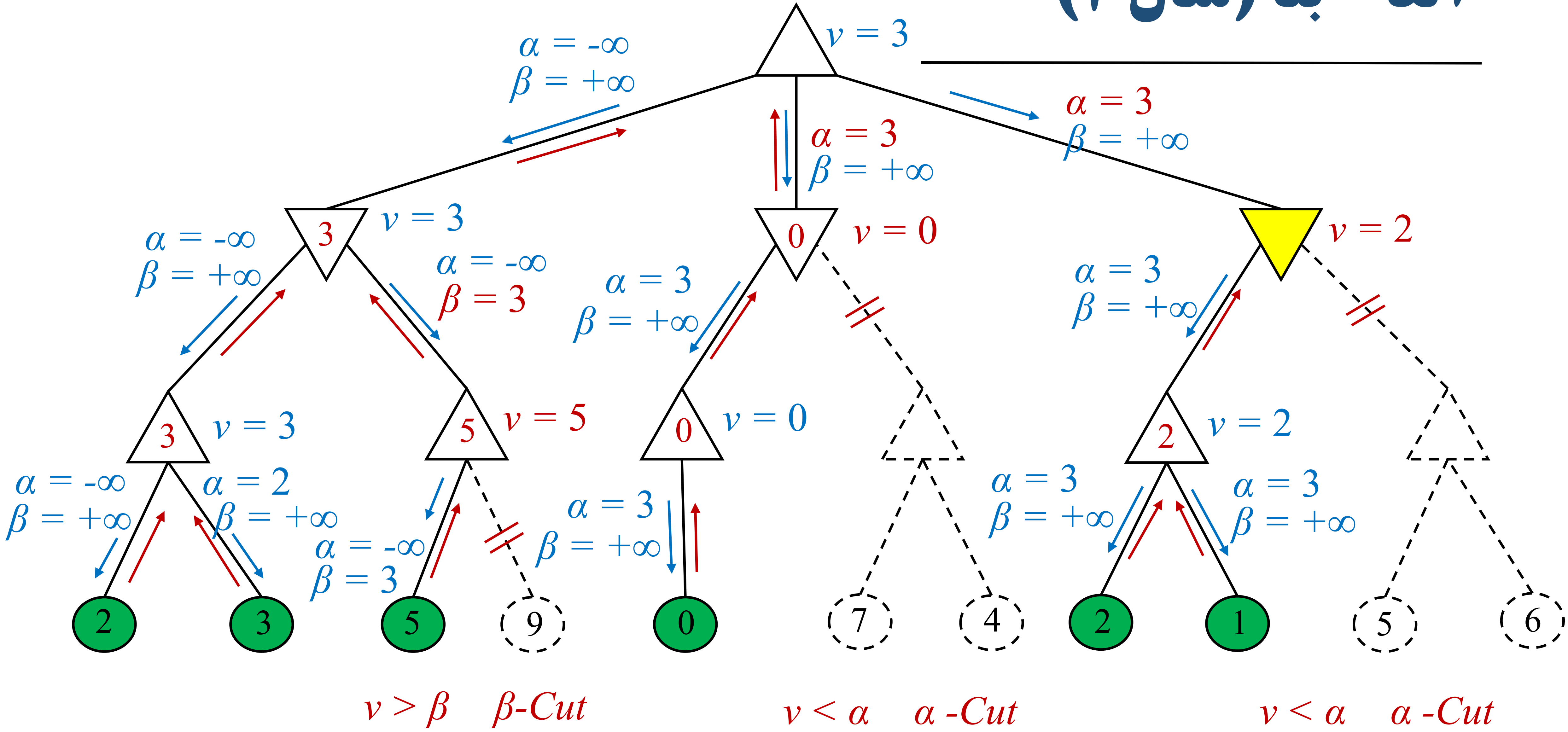
آلفا– بتا (مثال ۲)



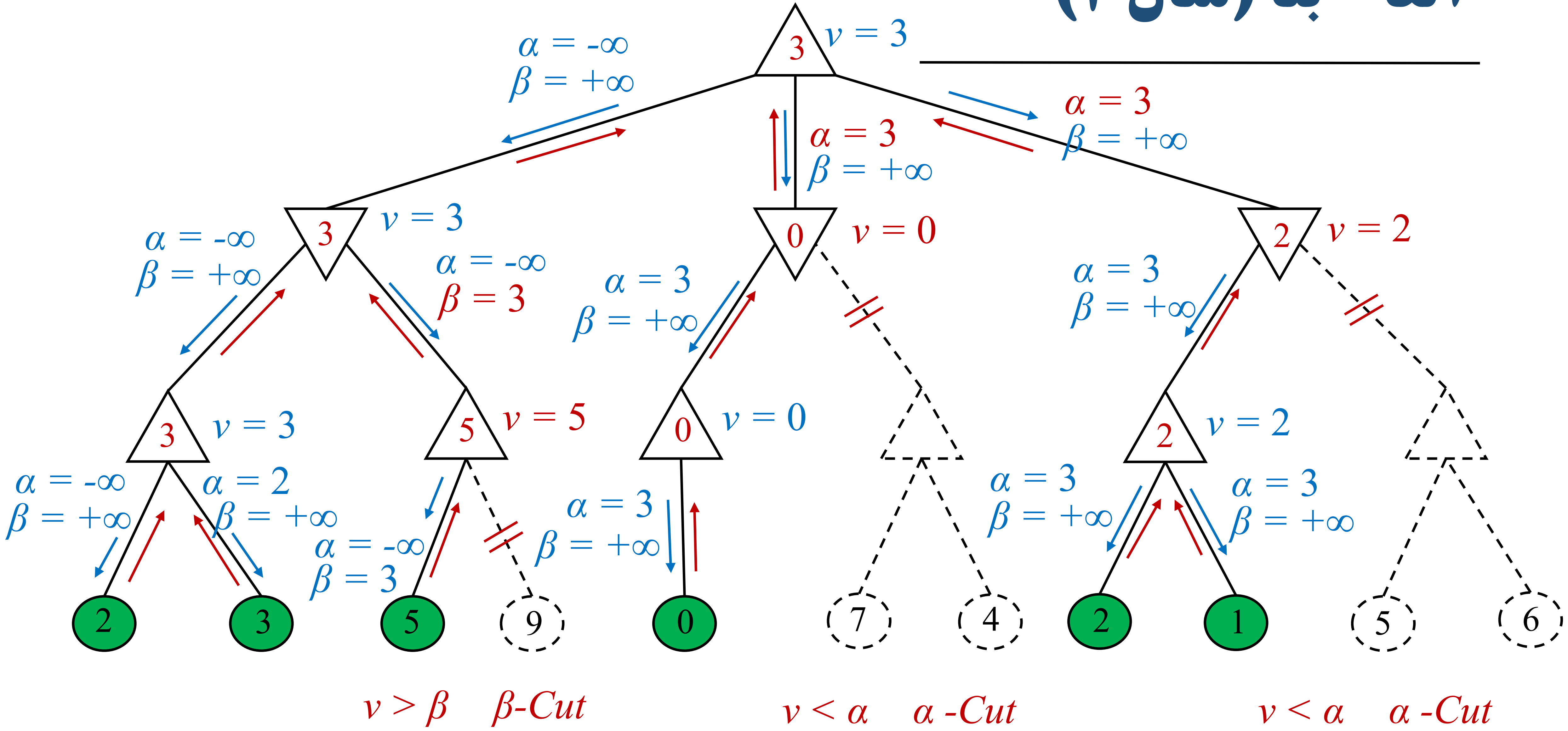
آلفا–بتا (مثال ۲)



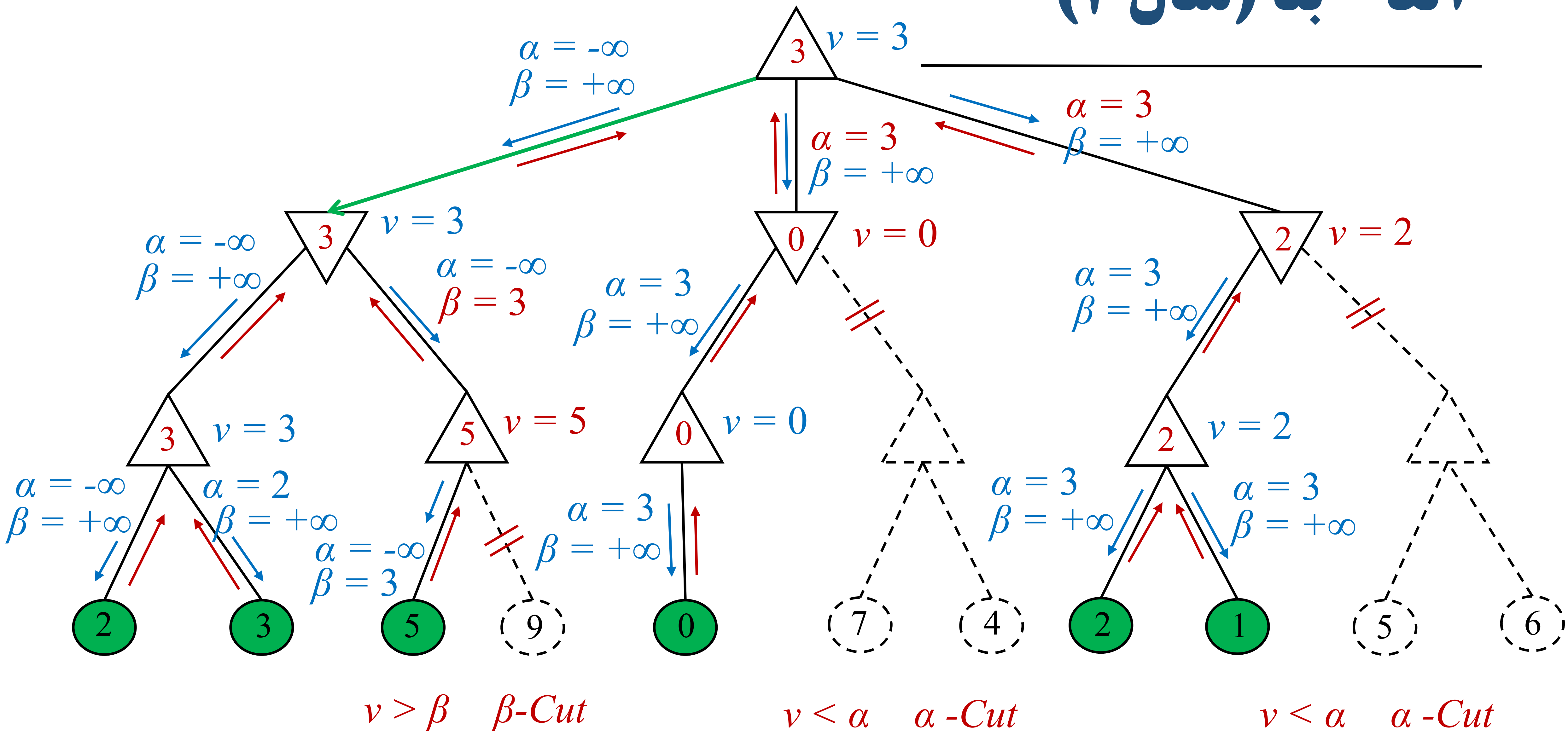
آلفا–بتا (مثال ۲)



آلفا–بتا (مثال ۲)

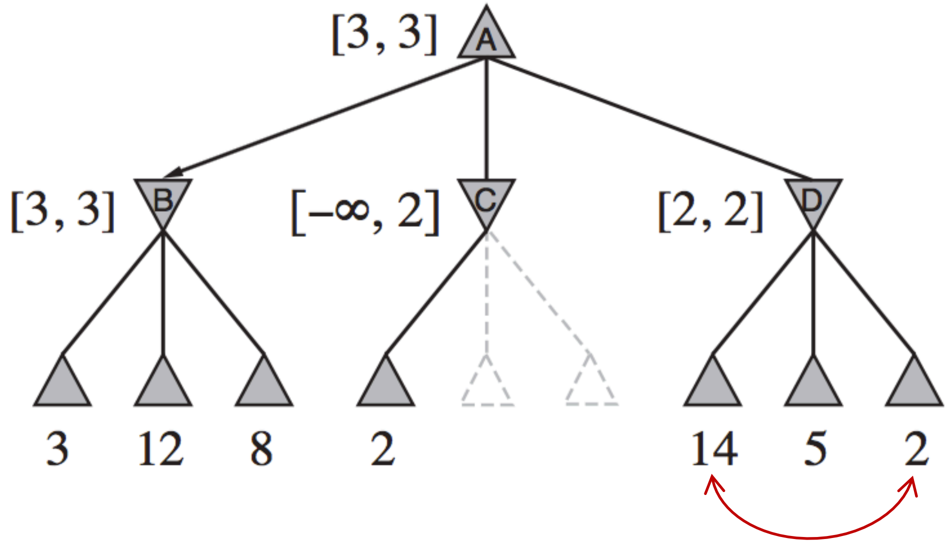


آلفا–بتا (مثال ۲)



ویژگی‌های الگوریتم آلفا–بتا

- ترتیب بررسی شاخه‌های درخت بر روی تعداد شاخه‌های هرس‌شده تاثیر می‌گذارد.
- برای بازی‌هایی که سودمندی گره‌ها از ∞- تا ∞+ است بیشترین هرس زمانی اتفاق می‌افتد که
  - برای هر گره MAX، فرزند با بیشترین سودمندی آن در سمت چپ‌ترین شاخه باشد.
  - برای هر گره MIN، فرزند با کم‌ترین سودمندی آن در سمت چپ‌ترین شاخه باشد.

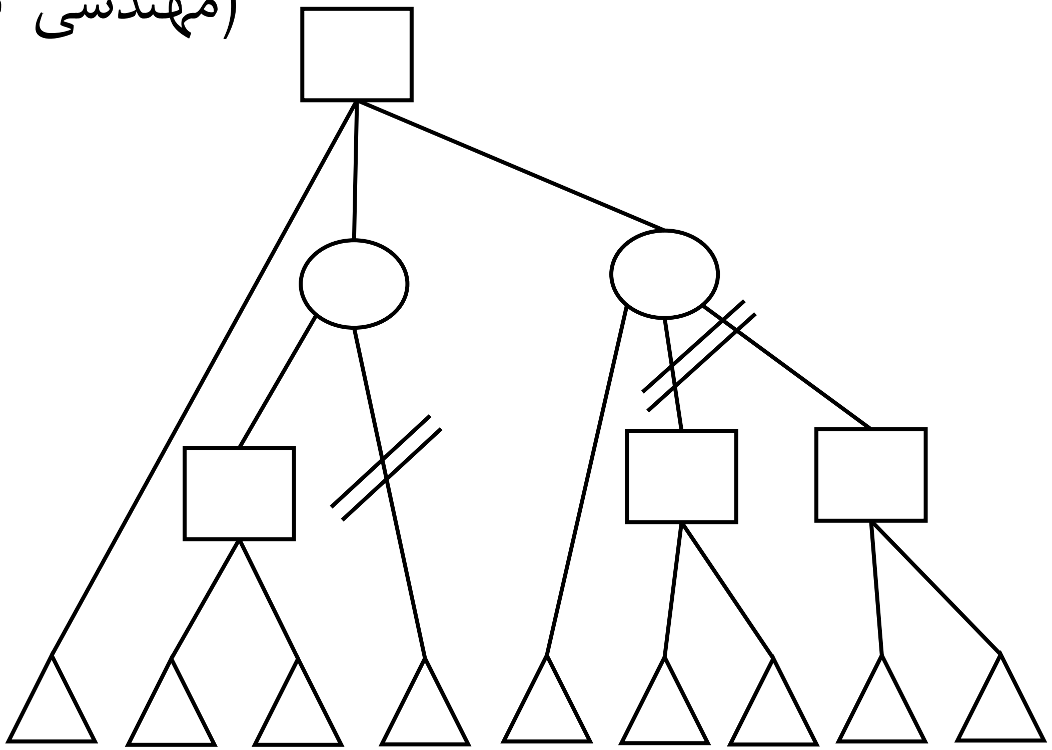


ویژگی‌های الگوریتم آلفا–بتا

- الگوریتم هرس آلفا– بتا در بهترین حالت می‌تواند حداکثر نصف شاخه‌های درخت را هرس کند.
  - در این‌صورت پیچیدگی زمانی در بهترین حالت برابر است با $O(b^{m/2})$
  - و اگر فاکتور انشعاب موثر برای MINIMAX برابر با b باشد برای آلفا–بتا برابر با $\sqrt{b}$ است.
  - سرعت الگوریتم هرس آلفا– بتا دو برابر الگوریتم بیشینه کمینه است. یعنی اگر در یک زمان معین، الگوریتم بیشینه کمینه تا عمق m از درخت را بررسی کند، الگوریتم هرس آلفا–بتا تا عمق 2m را بررسی خواهد کرد.
- اگر ترتیب بررسی شاخه‌ها و گره‌ها به‌طور تصادفی انتخاب شوند، تعداد کل گره‌هایی که بررسی می‌شوند، به‌طور متوسط برابر با $O(b^{3m/4})$ است.
- استفاده از جدول جابه‌جایی برای ذخیره‌سازی مقدار ارزیابی‌شده‌ی حالات تکراری

تست ۱

در گراف مقابل مربع نشانه بازیکن Max، دایره نشانه بازیکن Min و مثلث نشانه حالت پایانی است. اگر مقادیر ارزیابی بتوانند در فاصله بسته [0,10] باشند و با هرس آلفا–بتا فقط یال‌های علامت زده شده // حذف شوند، ترتیب گره‌های پایانی به‌ترتیب از چپ به راست در شکل کدام یک از گزینه‌های زیر خواهد بود؟ (مهندسی کامپیوتر ۹۰)



۱) ۴، ۱، ۰، ۱۰، ۹، ۳، ۵، ۲، ۸

۲) ۰، ۱، ۲، ۳، ۴، ۵، ۸، ۹، ۱۰

۳) ۴، ۲، ۱، ۰، ۸، ۱۰، ۳، ۵، ۹ ✓

۴) ۱۰، ۹، ۸، ۹، ۴، ۳، ۲، ۱، ۰

تصمیمات بی‌درنگ ناقص

- الگوریتم هرس آلفا–بتا نیاز دارد که برخی از شاخه‌ها را تا برگ‌ها پیمایش کند.
- در عمل گره‌های پایانه عمق زیادی دارند و رسیدن به آن‌ها زمان زیادی را می‌برد. در حالی‌که انتظار داریم الگوریتم تصمیم‌گیری در زمان معقولی پایان پذیرد.
- برای سریع‌تر شدن تصمیم‌گیری می‌توانیم درخت جست‌وجو را تا برگ‌ها پیمایش نکنیم بلکه آن را در عمق خاصی عمق برش (cutoff) پیمایش کنیم و سپس از روی سودمندی گره‌های آن عمق که احتمالا پایانه نیستند و انجام الگوریتم Minimax یا هرس آلفا–بتا تصمیم‌گیری کن یم.
  - چگونه می‌توان سودمندی گره‌های غیر پایانه را در عمق برش تعیین کرد؟؟
  - تا چه عمقی از درخت بازی را بهتر است پیمایش کنیم؟؟

تعیین سودمندی گره‌های غیر پایانه

- تعریف تابع ارزیابی(EVAL): تابعی است که تخمینی از سودمندی مورد انتظار یک وضعیت غیر پایانه بازی را برمی‌گرداند.

$$
\text{H-MINIMAX}(s,d) =
\begin{cases}
\text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\
\max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\
\min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}.
\end{cases}
$$

- ویژگی‌های تابع ارزیابی مطلوب:
  - باید حالت‌های پایانه را متناسب با تابع سودمندی ارزش‌دهی کند.
  - زمان لازم برای ارزیابی نباید زیاد شود.
  - در مورد حالات غیرپایانی، تابع ارزیابی باید وابستگی زیادی به شانس‌های واقعی بُرد داشته باشد.

تعریف مناسب یک تابع ارزیابی

- بیشتر توابع ارزیاب به وسیله محاسبه خصوصیات گوناگون حالت کار می‌کنند.
  - برای مثال تعداد سربازان در بازی شطرنج
- این خصوصیات در مجموع، طبقات یا دسته‌های مشابهی از حالات را تعریف می‌کنند.
  - حالاتی که در یک دسته قرار می‌گیرند، دارای مقادیر یکسانی برای تمامی خصوصیات می‌باشند.
- هر دسته مفروض شامل حالاتی خواهد بود که برخی از آن‌ها به بُرد، برخی به تساوی و برخی دیگر به شکست منجر می‌شوند.
- تابع ارزیاب قابلیت تشخیص این حالات را ندارد اما می‌تواند مقداری را که نشان‌گر نسبت حالات با هر خروجی است را تولید نماید.

تعریف مناسب یک تابع ارزیابی

- مثال: فرض کنید که براساس تجربه ما، از میان حالات یک دسته مشخص، ۷۲٪ از حالات به برد (۱)، ۲۰٪ به شکست (۰) و ۸٪ به تساوی (۱/۲) منجر شود:

$$(0.72 \times +1) + (0.20 \times 0) + (0.08 \times 1/2) = 0.76$$

- تابع ارزیابی نیازی ندارد ارزش گره‌ها را نزدیک به مقدار واقعی سودمندی آن‌ها تخمین بزند، بلکه تنها کافی است گره‌ها را همانند تابع سودمندی مرتب کند، یعنی به موقعیت‌های بهتر، ارزش بیشتری دهد.

- در عمل این نوع تحلیل و بررسی به دسته‌ها و بنابراین تجربه‌های بسیار زیادی به منظور تخمین تمام احتمال‌های برد نیاز دارد.

تعریف مناسب یک تابع ارزیابی

- بنابراین، برای تخمین سودمندی یک وضعیت غیر پایانه در یک بازی، برای آن بازی یک سری ویژگی در نظر می‌گیریم و به هرکدام یک ارزش نسبت می‌دهیم بعد از آن با ترکیب ارزش‌های ویژگی‌ها، سودمندی یک وضعیت را تخمین می‌زنیم.
- مثال:
  - f_i: تعداد هر نوع قطعه در صفحه
  - w_i: ارزش آن قطعات (۱ برای پیاده، ۳ برای اسب یا فیل،۵ برای رخ و ...)

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- ارزش دو فیل از دو برابر ارزش یک فیل بیشتر است یا این‌که ارزش یک فیل در انتهای بازی بیشتر می‌شود. بنابراین بهتر است که یک تابع غیرخطی از ویژگی‌ها را به عنوان تابع ارزیابی استفاده کرد.

پایان دادن به جستجو

- در زمان مناسب جستجو را قطع و تابع هیوریستیک EVAL را فراخوانی می‌کند.

if CUTOFF-TEST(state, depth) then return EVAL(state)

- جستجوی عمقی با حداکثر عمق محدود:
  - درخت جستجو را تا عمق مشخص d پیمایش کنیم.
  - مشکل در تعیین عمق d: اگر d بزرگ باشد تصمیم دقیق‌تری می‌گیرد اما ممکن است از زمان قابل قبول فراتر رود. برعکس اگر d کوچک باشد الگوریتم سریع تصمیم می‌گیرد ولی از کل زمان مجاز خود استفاده نمی‌کند.
- استفاده از جستجوی عمقی تکراری (IDS):
  - از این جستجو استفاده می‌کنیم و تا اتمام زمان مجاز جستجو را ادامه می‌دهیم.
  - در این صورت تا جای ممکن عمق بیشتری از درخت را دیده‌ایم و تصمیم دقیق‌تری می‌گیریم.

پایان دادن به جستجو

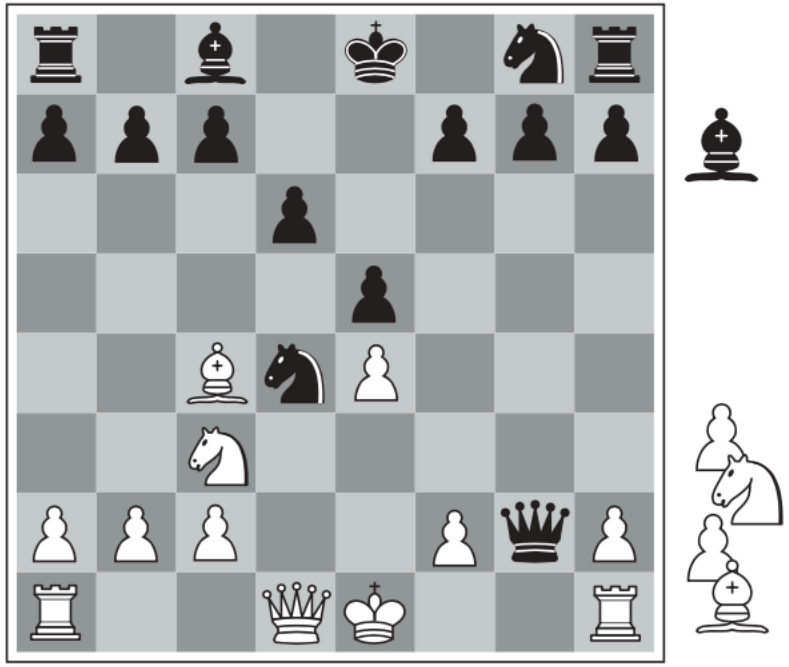
- هر دو روش گفته شده در اسلاید قبل به دلیل غیر دقیق بودن تابع ارزیابی ممکن است دچار مشکلاتی شوند:

۱- وضعیت غیر ساکن

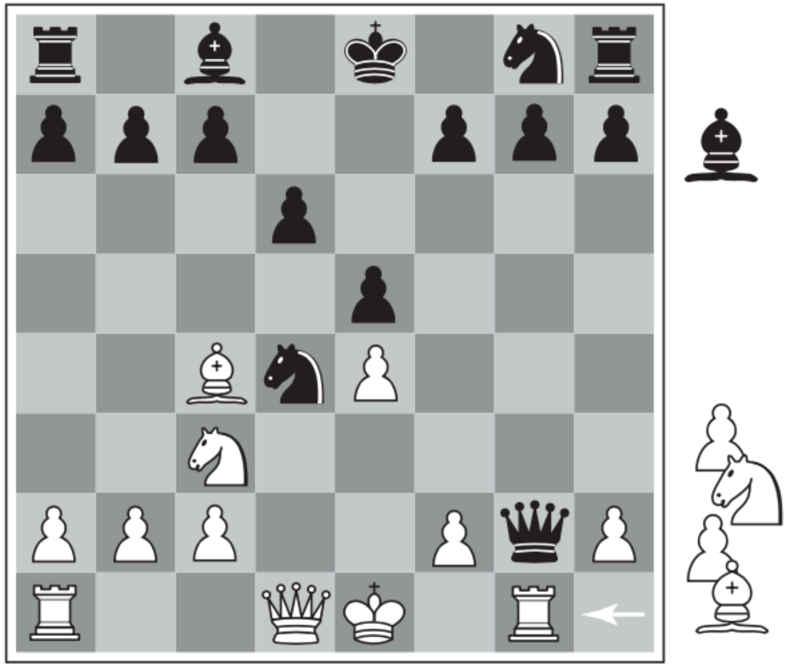
۲- اثر افق

مشکل وضعیت‌های غیرساکن

- وضعیت ساکن: وضعیتی است که به احتمال زیاد در آینده نزدیک تغییر چندانی نمی‌کند. چه فرزندانش بررسی شوند یا نه ارزش تخمینی آن تغییر چندانی نمی‌کند و جانشین‌های آن تا چند مرحله جلوتر ارزشی نزدیک به ارزش آن داشته باشند.



(a) White to move



(b) White to move

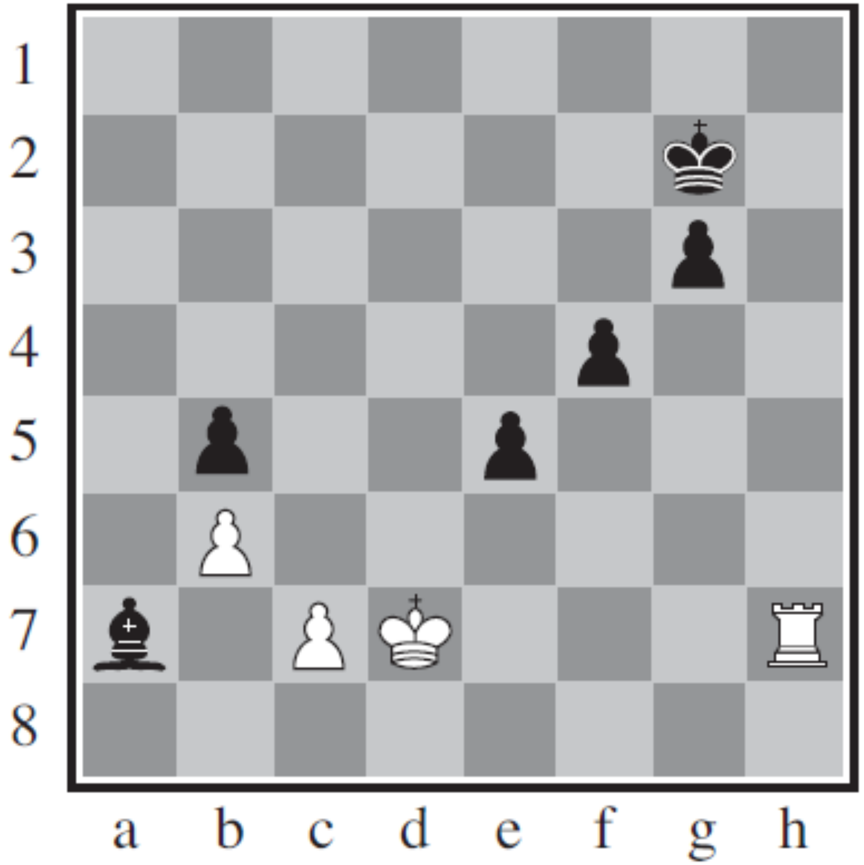
مشکل وضعیت‌های غیرساکن

- این وضعیت‌ها زمانی مشکل‌ساز هستند که یک گره در عمق برش، متناظر با یک وضعیت غیرساکن باشد. در این حالت چون سودمندی این گره توسط تابع ارزیابی غیردقیق تخمین زده می‌شود، تصمیم‌گیری صورت گرفته نیز غیردقیق خواهد بود.

- برای حل این مشکل باید تابع ارزیابی را فقط به وضعیت‌های ساکن اعمال کرد. وضعیت‌هایی که ساکن نیستند را می‌توان بسط داد تا به وضعیت‌های ساکن برسیم و سپس آن وضعیت‌های ساکن را ارزیابی کرد. به این کار جستجوی سکون گویند.

مشکل اثر افق

- وقتی به‌وجود می‌آید که برنامه با حرکتی از سوی رقیب مواجه شود که اثرات مخرب زیادی در پی دارد و قابل اجتناب نیست اما می‌توان آن را به تاخیر انداخت.



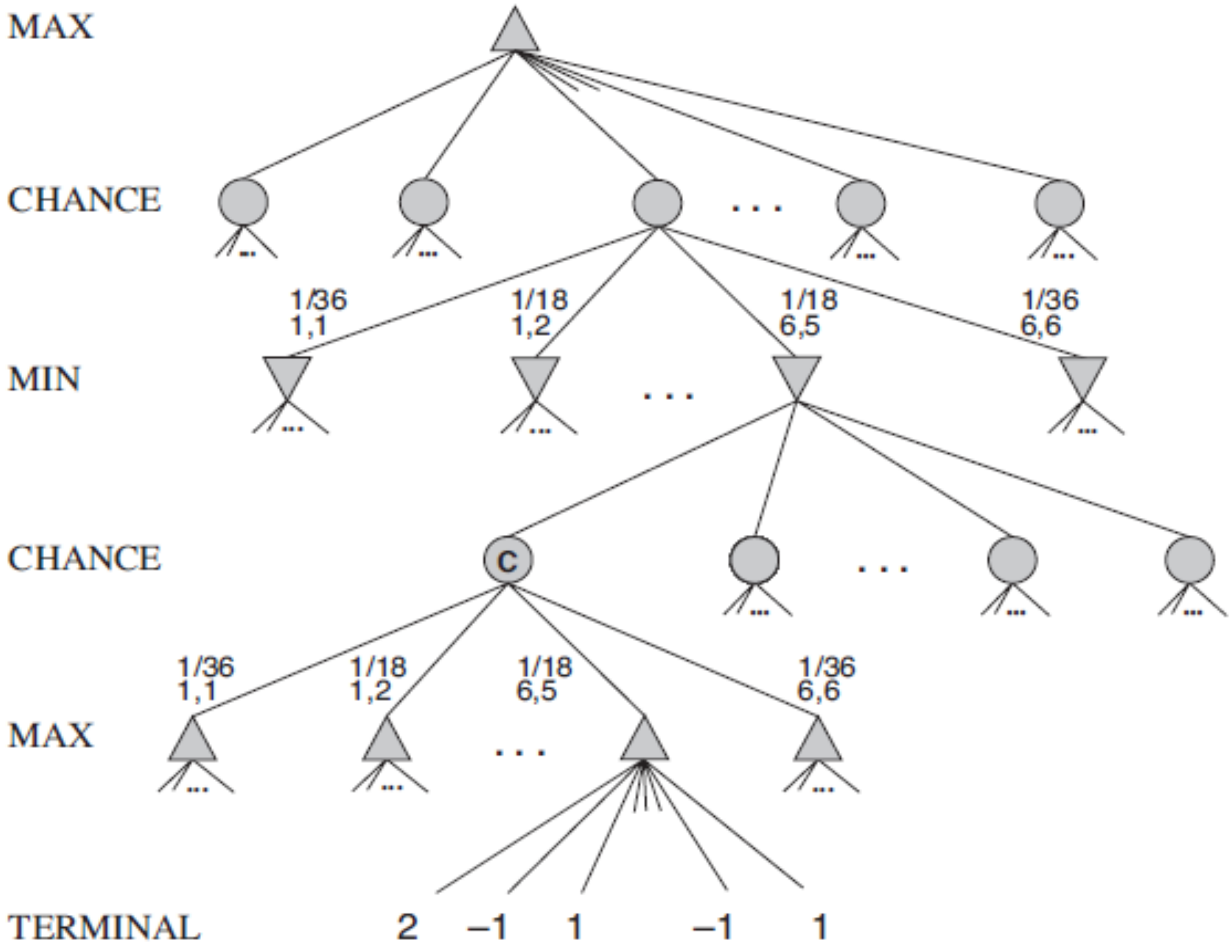
کم کردن مشکل اثر افق

- استفاده از سخت‌افزار قوی‌تر:
  - در این صورت می‌توان جستجو را تا عمق بیشتری ادامه داد.
- استفاده از بسط‌های یکتا (Singular Extension):
  - به حرکتی گفته می‌شود که از تمام حرکت‌های ممکن دیگر در یک وضعیت خاص بهتر باشد. اگر بدانیم یکی از فرزندان یک گره از بقیه فرزندانش بهتر است. فقط آن فرزند را بررسی می‌کنیم و بسط می‌دهیم.
  - اما به دلیل غیردقیق بودن تابع ارزیابی، برای اطمینان از این که یک فرزند واقعا از بقیه بهتر است هرگاه در جایی از درخت، یک حرکت با توجه به تابع ارزیابی بهتر از سایر حرکت‌ها بود آن حرکت را به خاطر می سپاریم و جستجو را به‌طور معمول ادامه می‌دهیم. وقتی جستجو به عمق برش رسید الگوریتم با توجه به ارزش گره‌های عمق برش بررسی می‌کند که آیا آن حرکت هنوز هم یک حرکت خوب است یا نه. اگر خوب باشد، این بار فقط شاخه مربوط به آن حرکت را تا عمق بیشتری بررسی می‌کنیم.

هرس کردن پیش‌رو

- منظور از هرس کردن پیش‌رو این است که بعضی از حرکت‌ها در یک گره فوراً و بدون بررسی حذف می‌شوند. مثلا می‌توان گره‌هایی که طبق تابع ارزیابی مناسب نیستند را دیگر ادامه نداد.
- یک روش انجام هرس پیشرو جست‌وجوی پرتو است یعنی در هر لایه به‌جای در نظر گرفتن تمام حرکت‌های ممکن فقط «پرتوی» از n بهترین حرکت را در نظر می‌گیریم.
  - متاسفانه این روش بسیار خطرناک است زیرا هیچ تضمینی وجود ندارد که بهترین حرکت هرس نشود.
  - هرس کردن پیش‌رو بهتر است برای گره‌های در عمق زیاد از درخت جستجو انجام شود. اگر این کار در نزدیکی ریشه انجام شود می‌تواند فاجعه‌انگیز باشد چون الگوریتم بسیاری از حرکت‌های بدیهی را از دست خواهد داد.

بازی های حاوی عنصر شانس



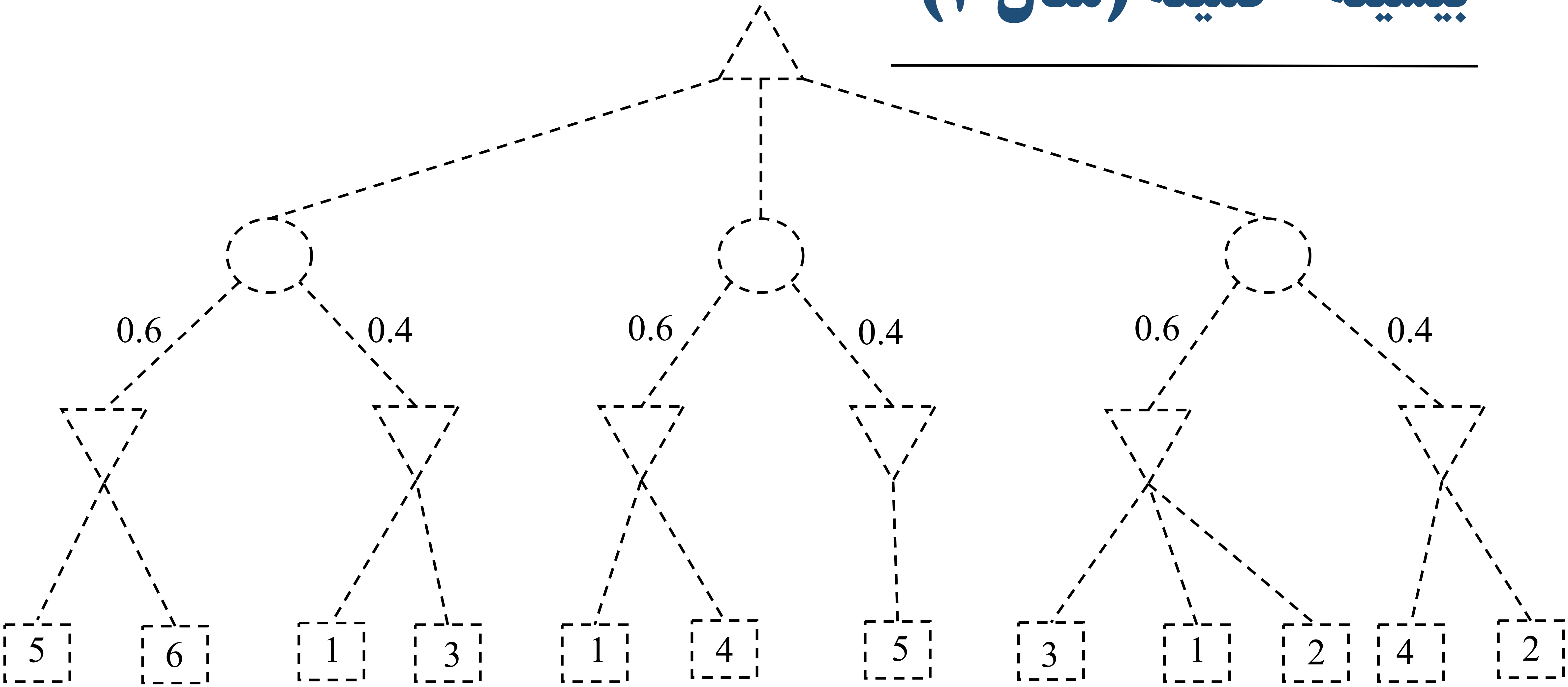
- در بسیاری از بازی‌ها یک عنصر شانس مانند تاس یا سکه وجود دارد که بازیکن‌ها به نوبت از آن استفاده می‌کنند. این عنصر شانس باعث می‌شود وضعیت‌های ممکن در آینده برای بازیکن‌ها قابل پیش‌بینی نباشد.
- در درخت مقابل بازیکن Max می‌خواهد تصمیم بگیرد کدام یک از اعمال ممکن را انجام دهد.
- توجه: گره‌های شانس در سطح یک مربوط به استفاده بازیکن Min از عنصر شانس است.

الگوریتم بیشینه‌کمینه برای بازی های حاوی عنصر شانس

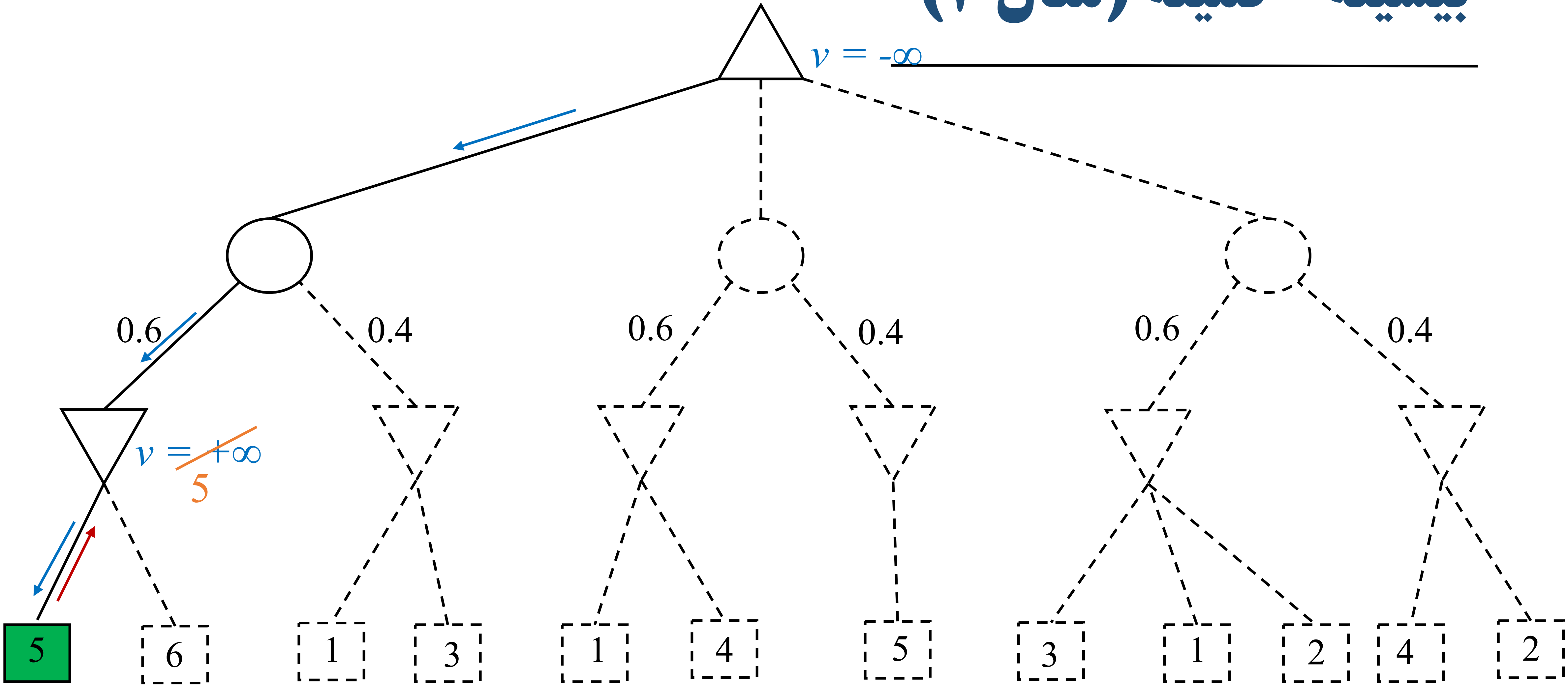
- ما می‌خواهیم حرکتی را انتخاب کنیم که بهترین موقعیت را ایجاد کند اما فاقد MINMAX متناهی هستیم. در عوض می‌توانیم مقدار مورد انتظار موقعیت را شناسایی کنیم که میانگین تمام نتایج ممکن مربوط به گره‌های شانس است.
- به این ترتیب می‌توان مقدار MIINMAX مربوط به بازی‌های قطعی را به یک مقدار MINMAX مورد انتظار برای بازی‌هایی با گره شانس تعمیم داد.

$$
\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \\ \sum_r P(r)\textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{Player}(s) = \textsc{chance} \end{cases}
$$

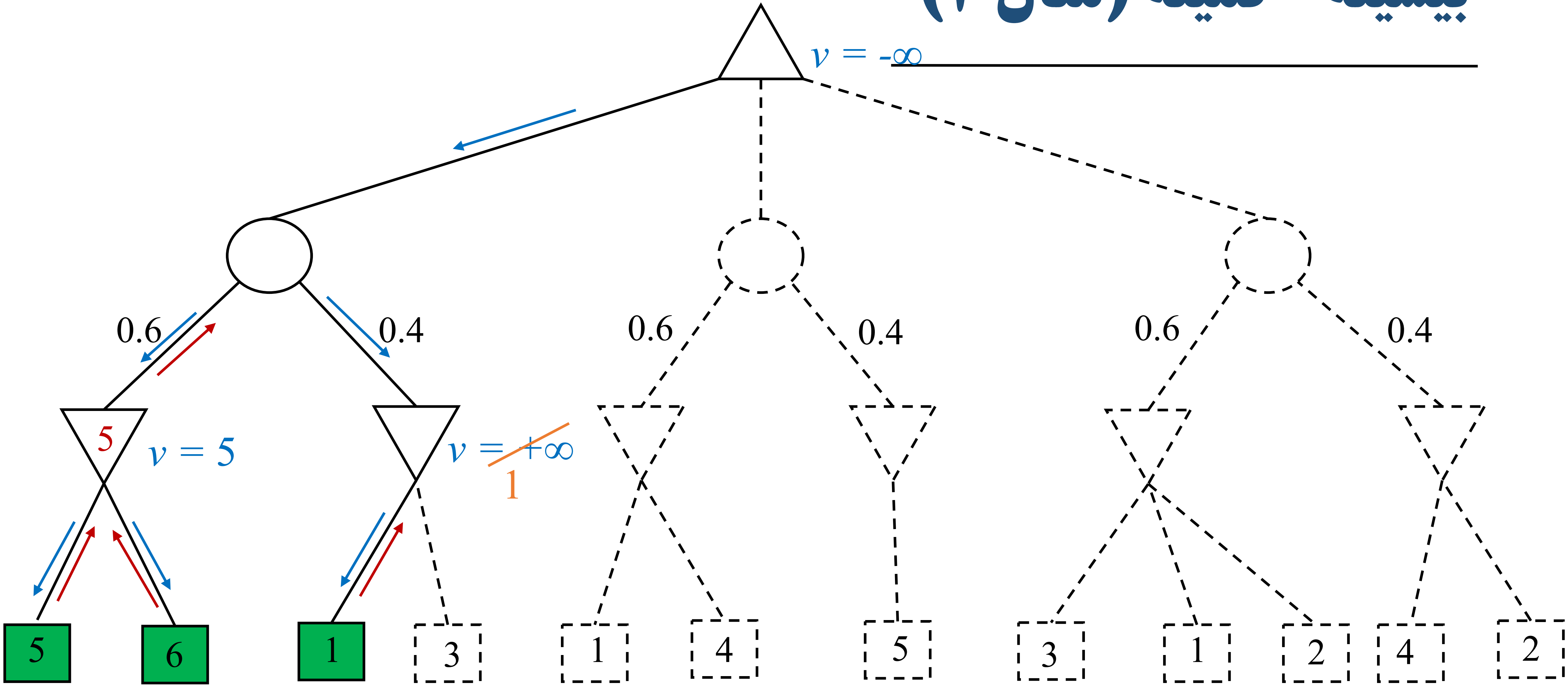
بیشینه–کمینه (مثال ۲)



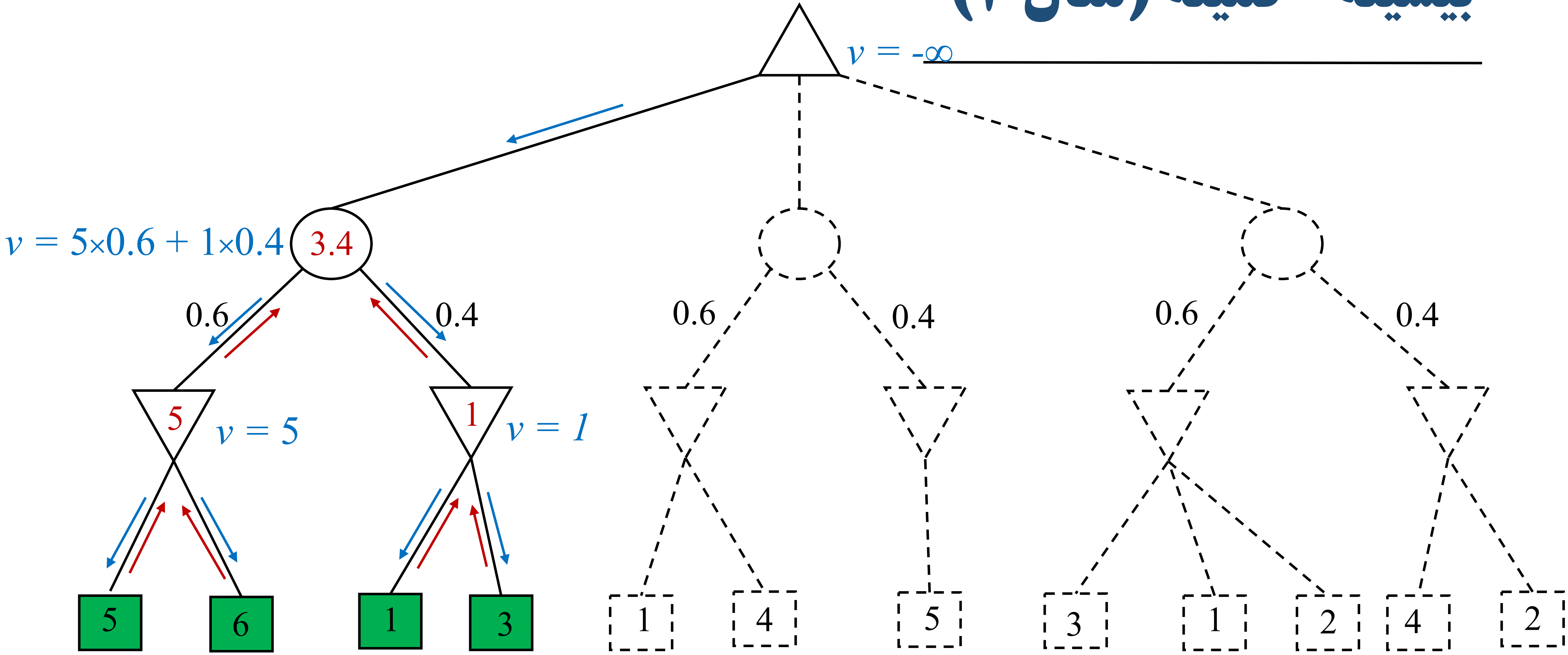
بیشینه–کمینه (مثال ۲)



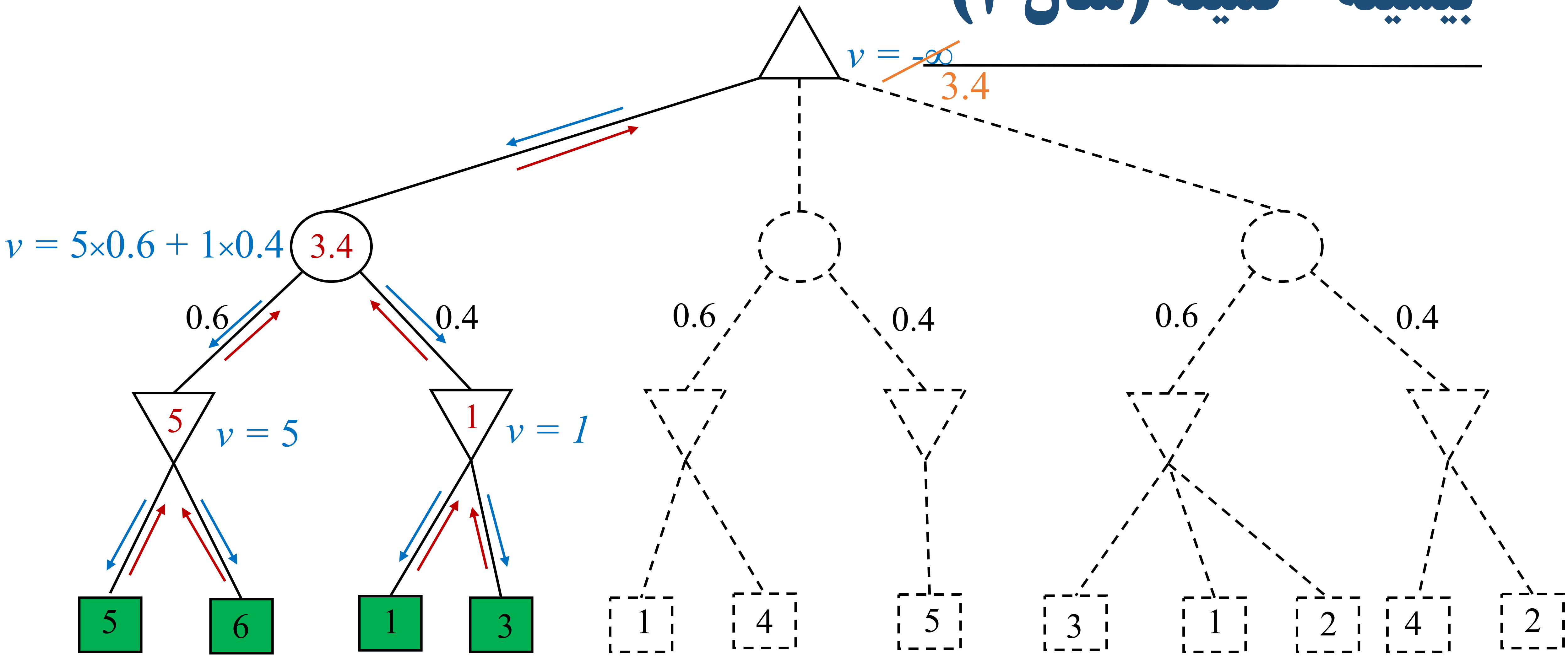
بیشینه–کمینه (مثال ۲)



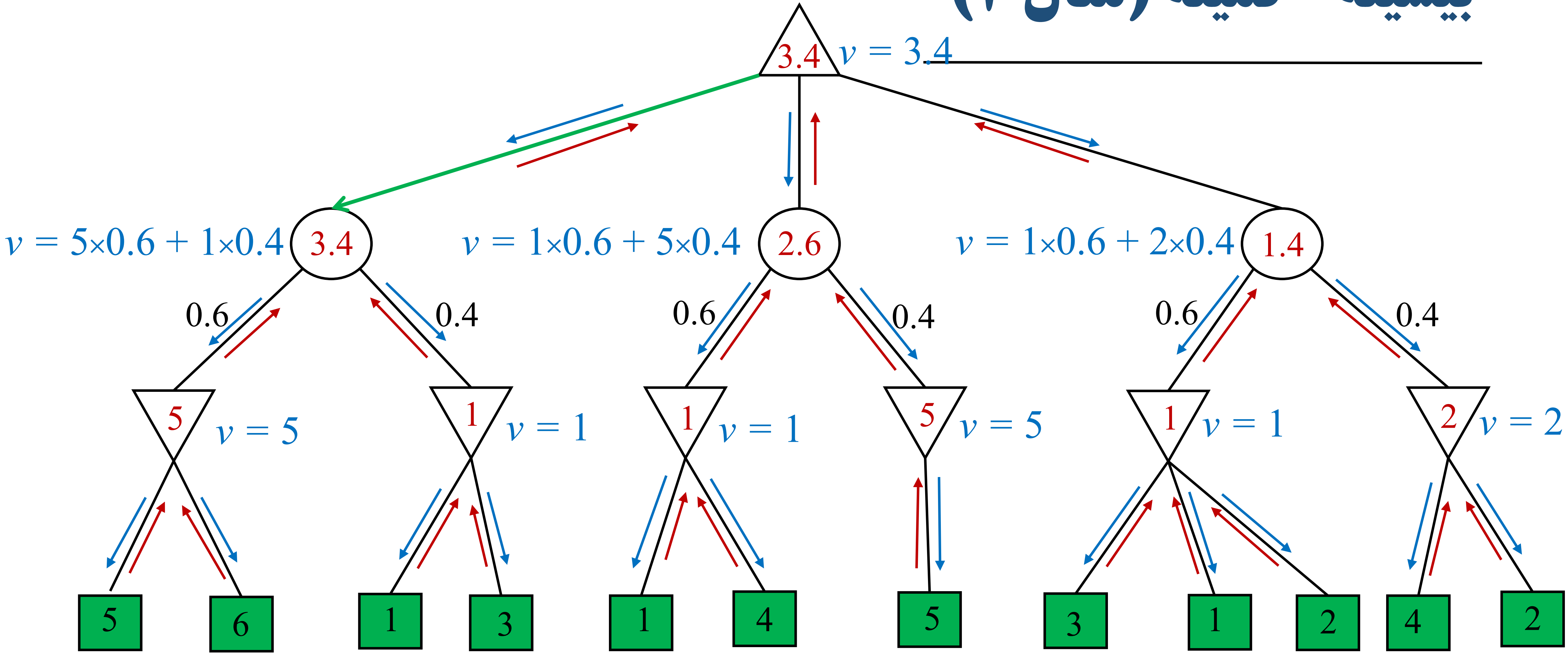
بیشینه–کمینه (مثال ۲)



بیشینه–کمینه (مثال ۲)



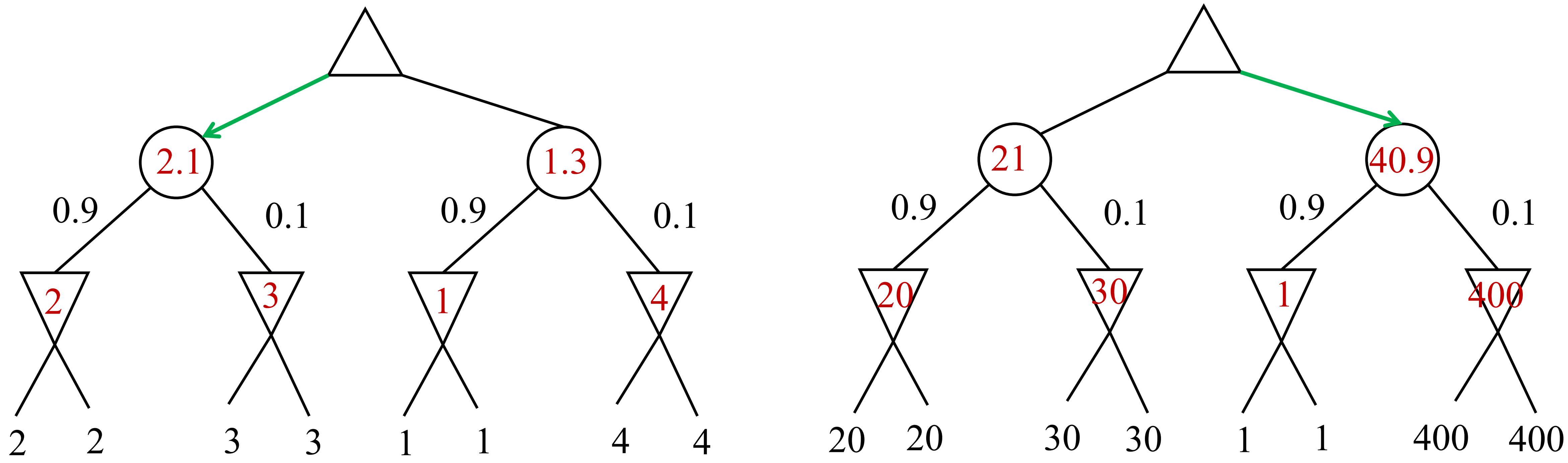
بیشینه–کمینه (مثال ۲)



الگوریتم بیشینه‌کمینه برای بازی های حاوی عنصر شانس

- همانند بازی‌های بدون عنصر شانس، اگر بررسی هر شاخه تا برگ‌ها زمان‌بر باشد، می‌توان درخت را تا عمق خاصی پیمایش کرد و از تابع ارزیابی برای ارزیابی تخمینی گره‌های غیرپایانه استفاده کرد.
  - در بازی‌های فاقد عنصر شانس، لازم نبود تابع ارزیابی اررزش گره‌ها را نزدیک به مقدار واقعی سودمندی آن‌ها تخمین بزند، بلکه تنها کافی بود گره‌ها را به‌ترتیب ارجح بودنشان ارزش‌دهی و مرتب کند. اما در بازی‌های حاوی عنصر شانس، چنین شرطی کافی نیست.
  - تابع ارزیابی در این بازی‌ها باید یک جابه‌جایی خطی مثبت از سودمندی واقعی گره‌ها باشد. یعنی مقدار x تبدیل به $ax+b$ $(a>0)$ شود.
  - مثال بعد نشان می‌دهد که تغییر با حفظ ترتیب در مقادیر برگ‌ها در بازی‌های حاوی عنصر شانس، بهترین حرکت را ممکن است عوض کند.

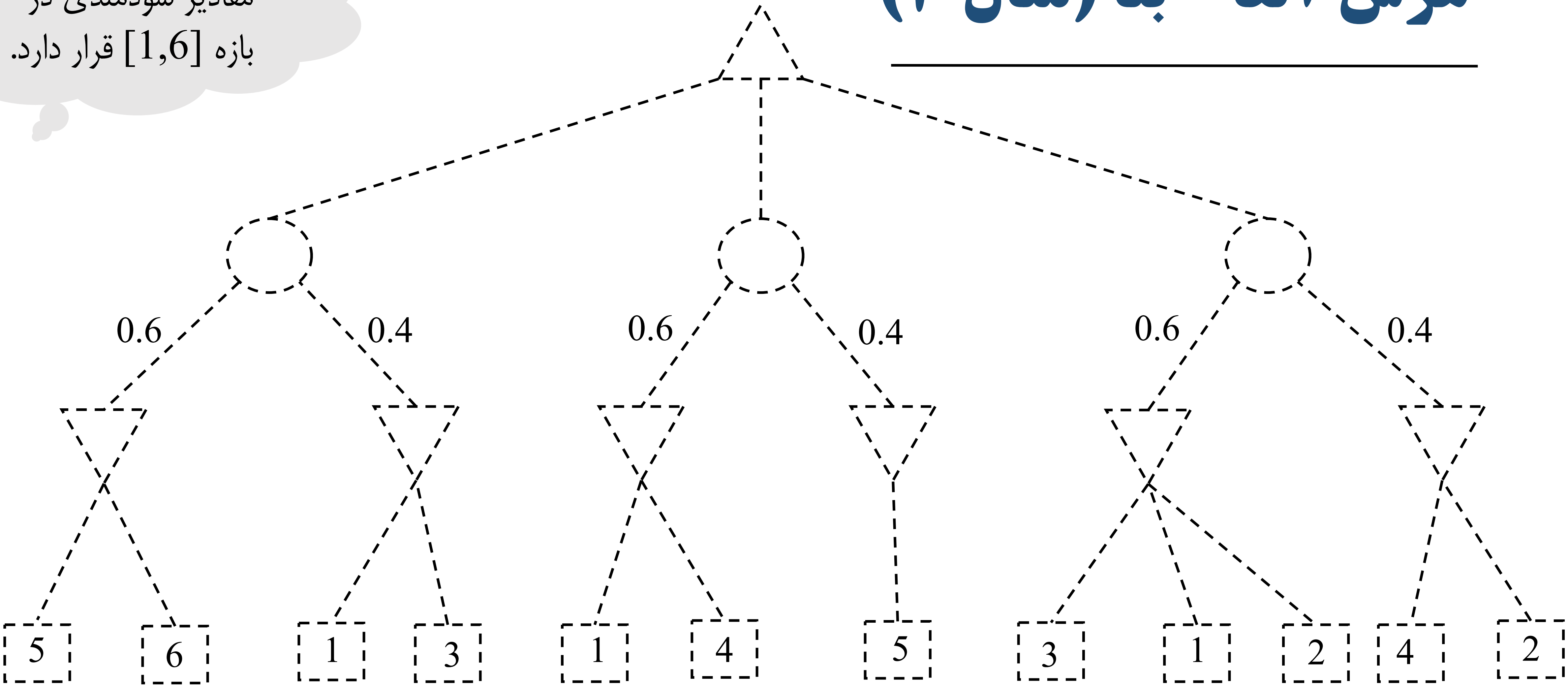
الگوریتم بیشینه‌کمینه برای بازی های حاوی عنصر شانس



- اگر عنصر شانس، n مقدار مختلف داشته باشد، پیچیدگی زمانی $O(b^m n^m)$ خواهد بود. در این صورت، حداکثر عمق قابل دسترسی (با توجه به محدودیت زمانی) کمتر از بازی‌های بدون عنصر شانس خواهد بود.

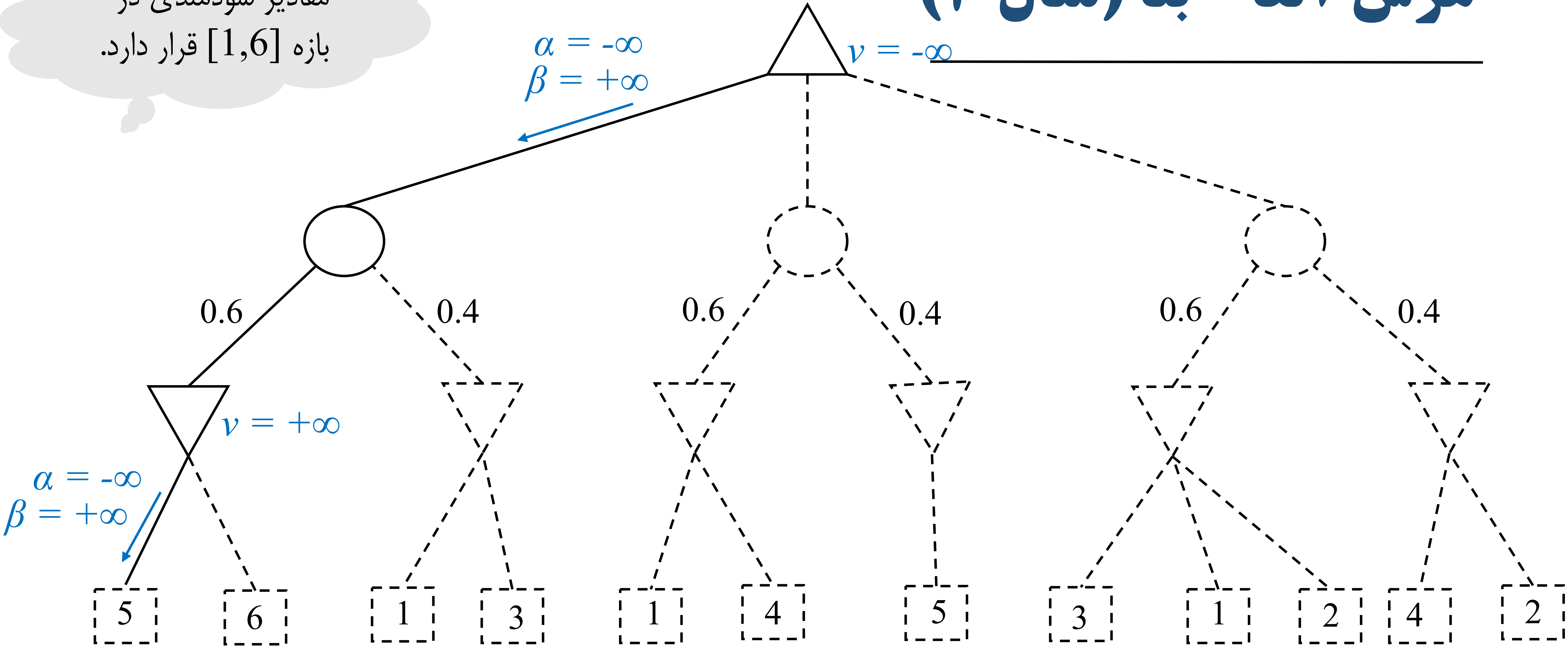
هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در
بازه [1,6] قرار دارد.



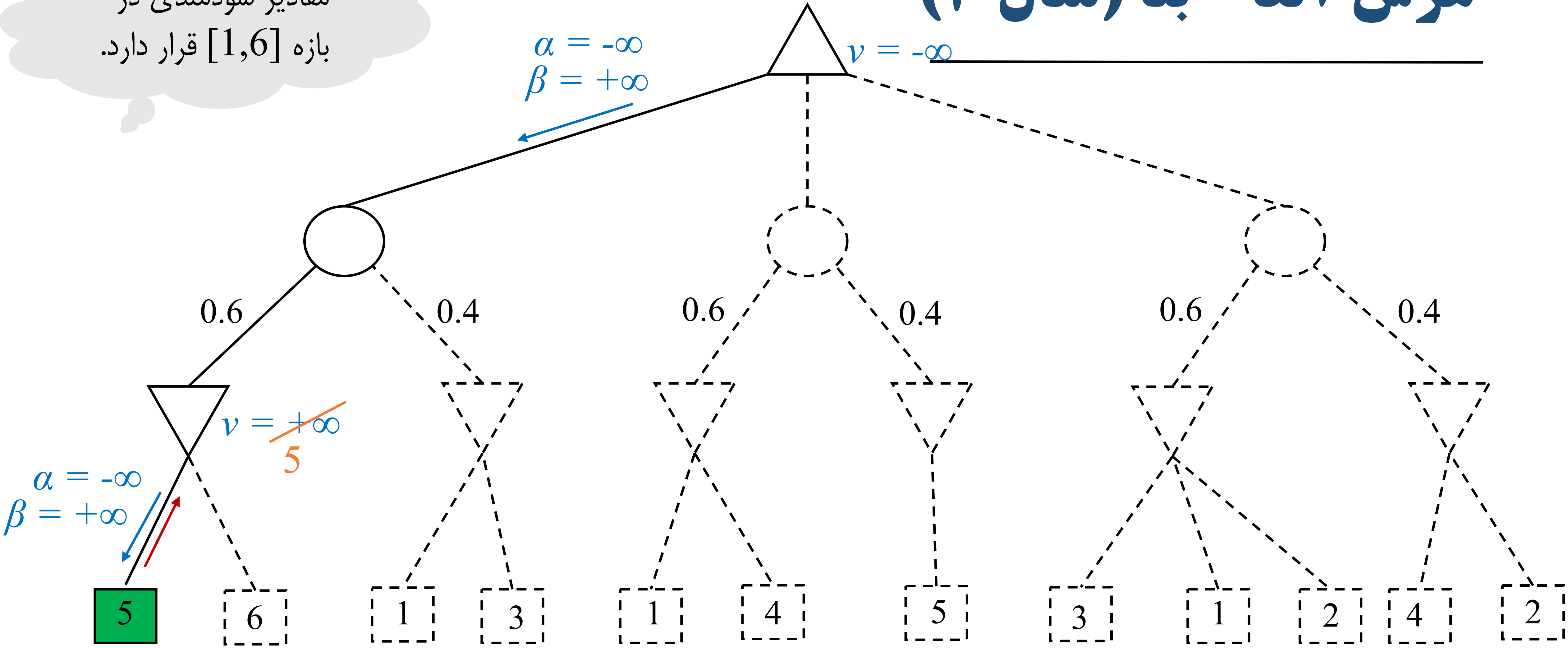
هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در
بازه [1,6] قرار دارد.



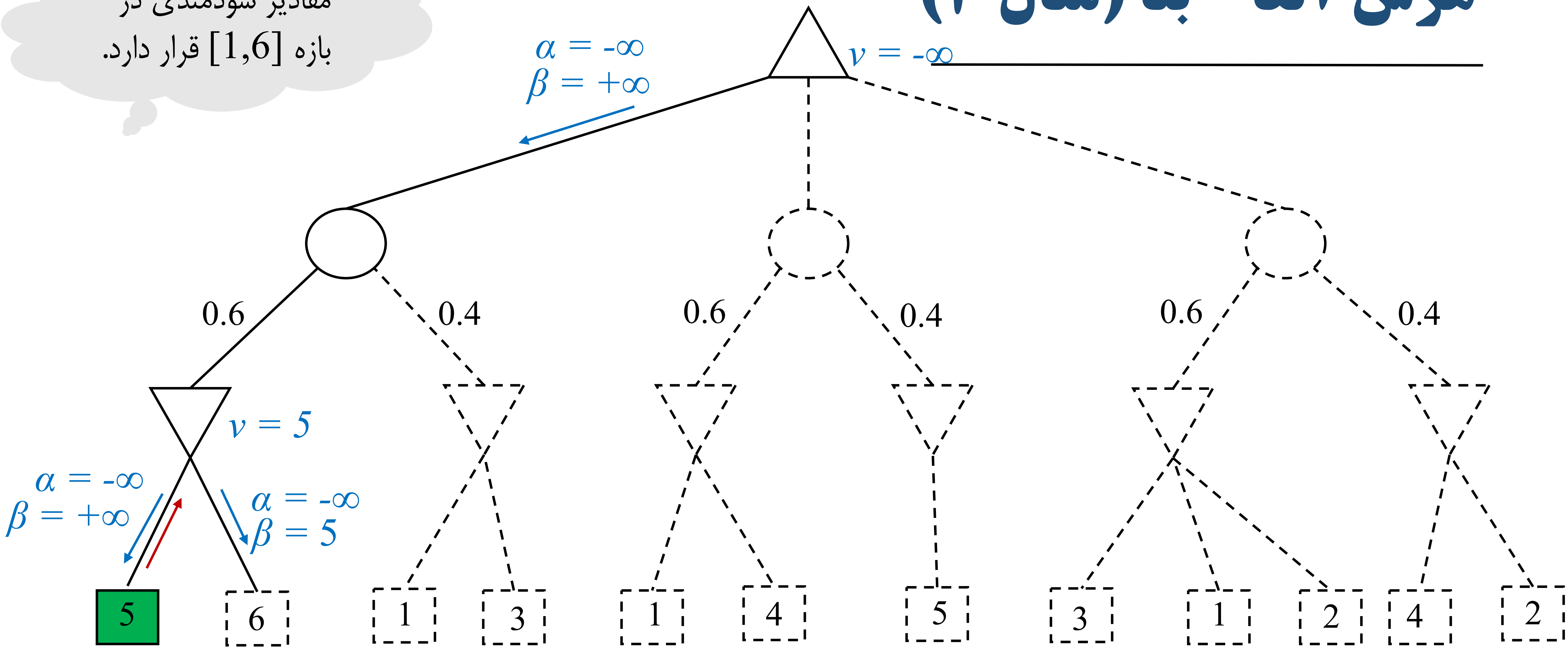
هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در
بازه [1,6] قرار دارد.



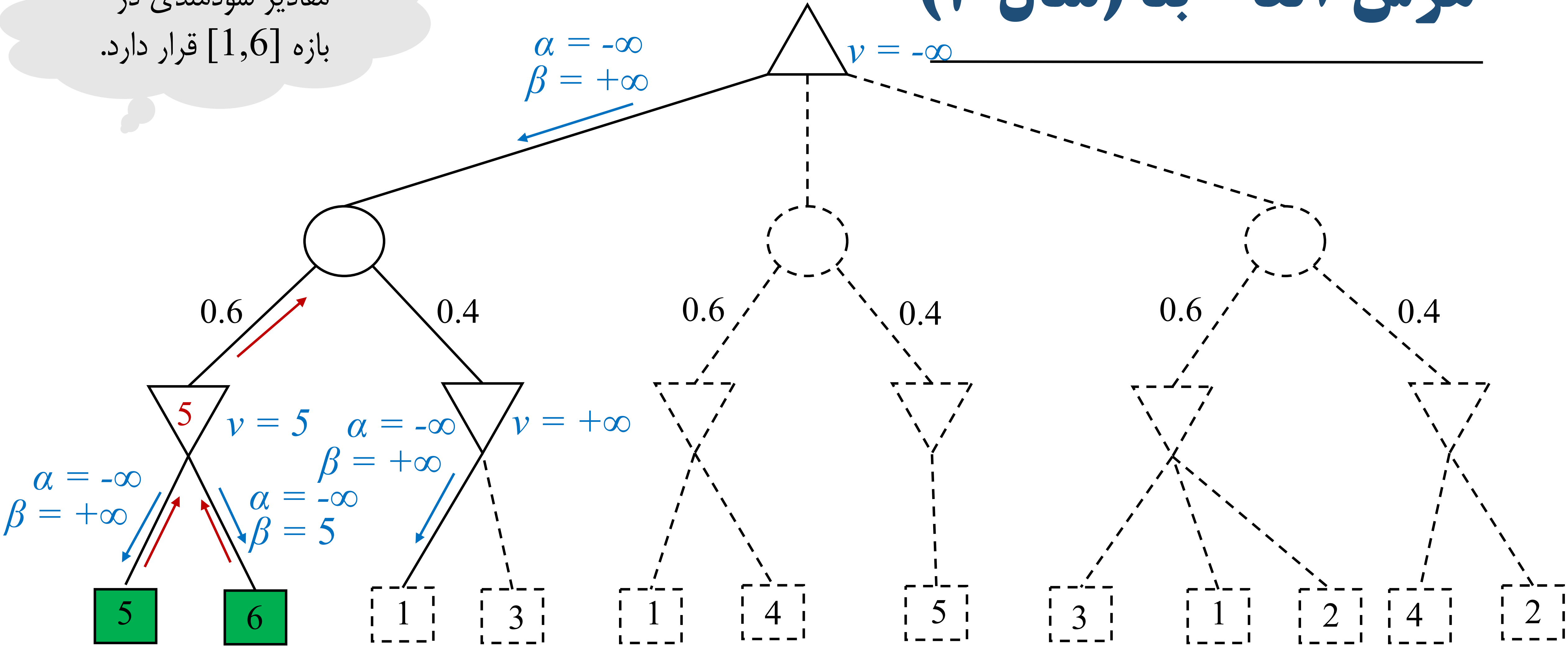
هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در
بازه [1,6] قرار دارد.



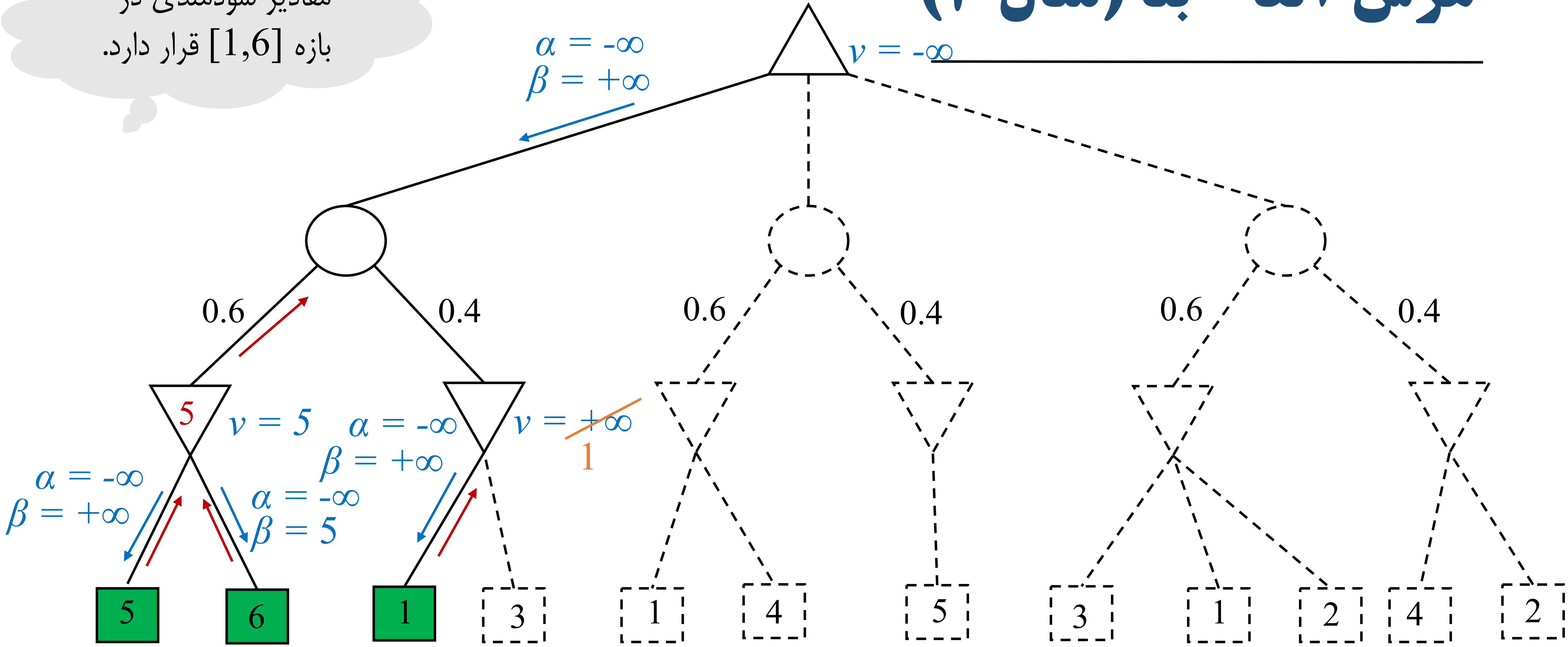
هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در
بازه [1,6] قرار دارد.

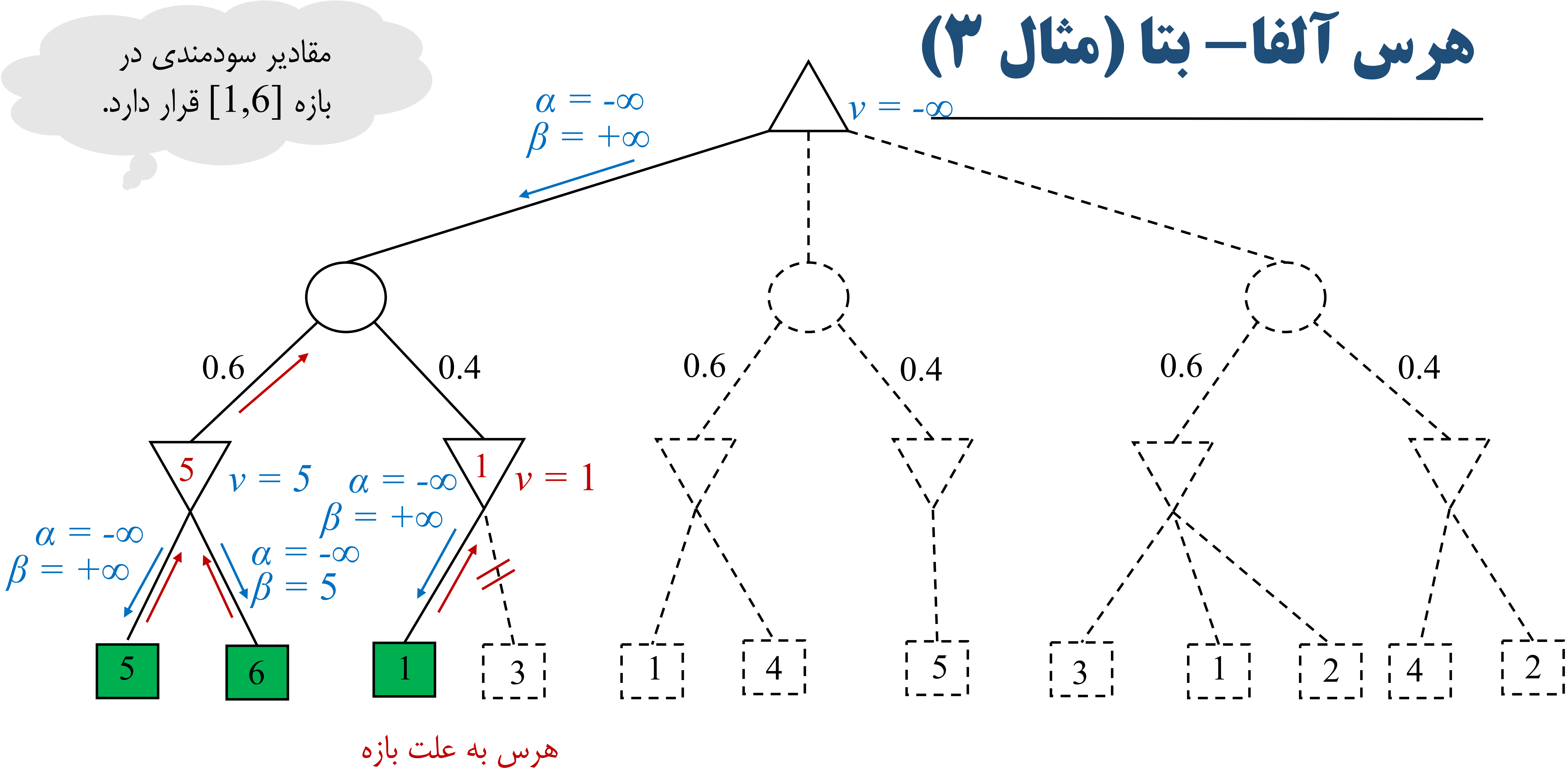


هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در
بازه [1,6] قرار دارد.

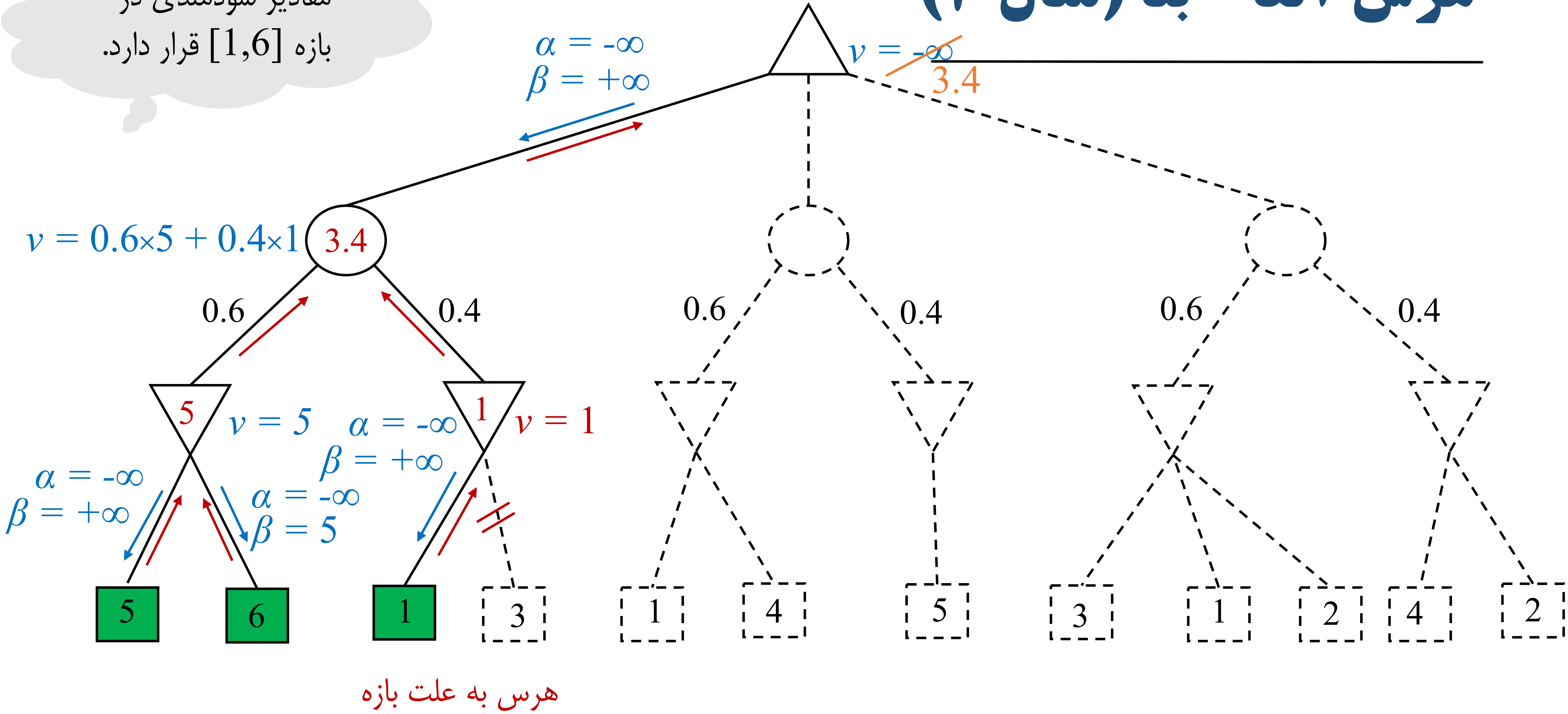


هرس آلفا– بتا (مثال ۳)



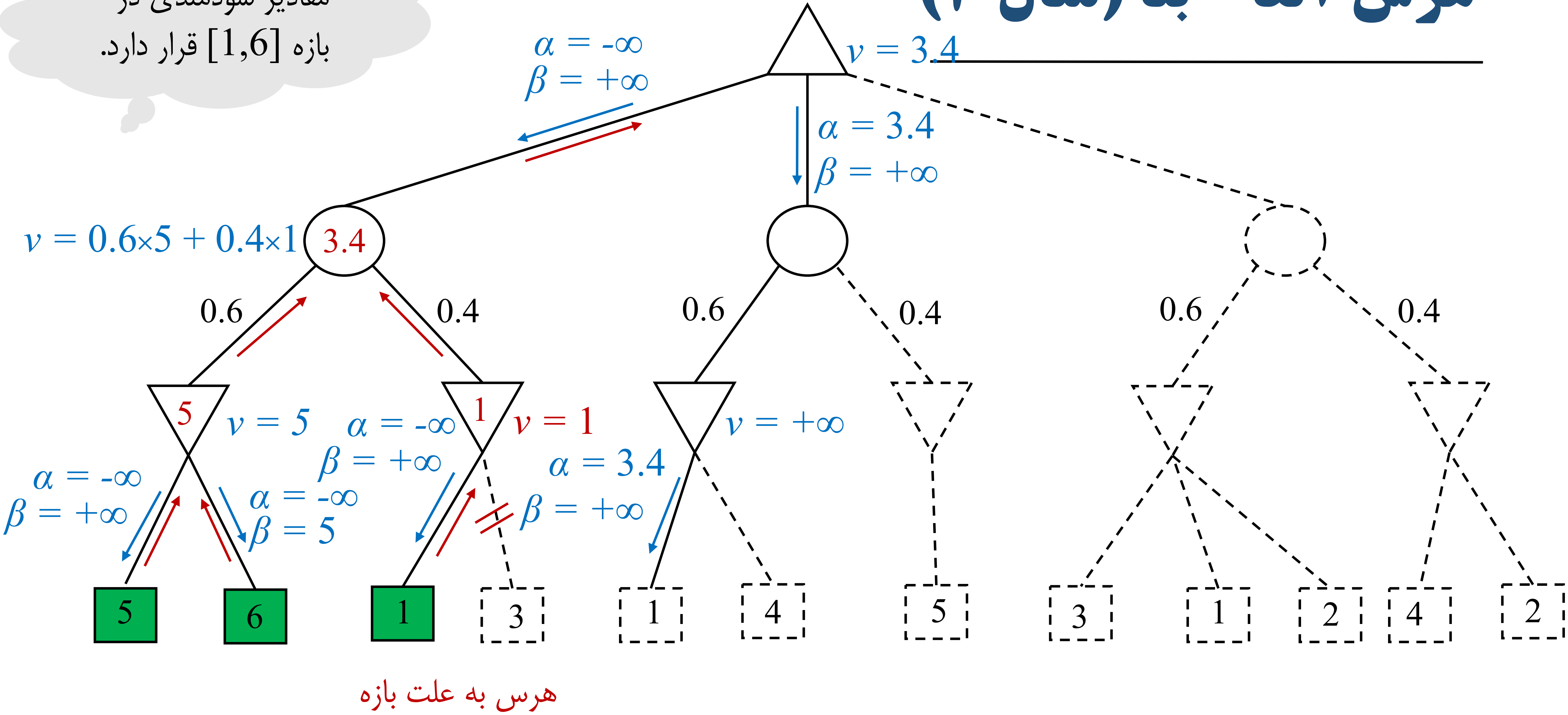
هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در
بازه [1,6] قرار دارد.



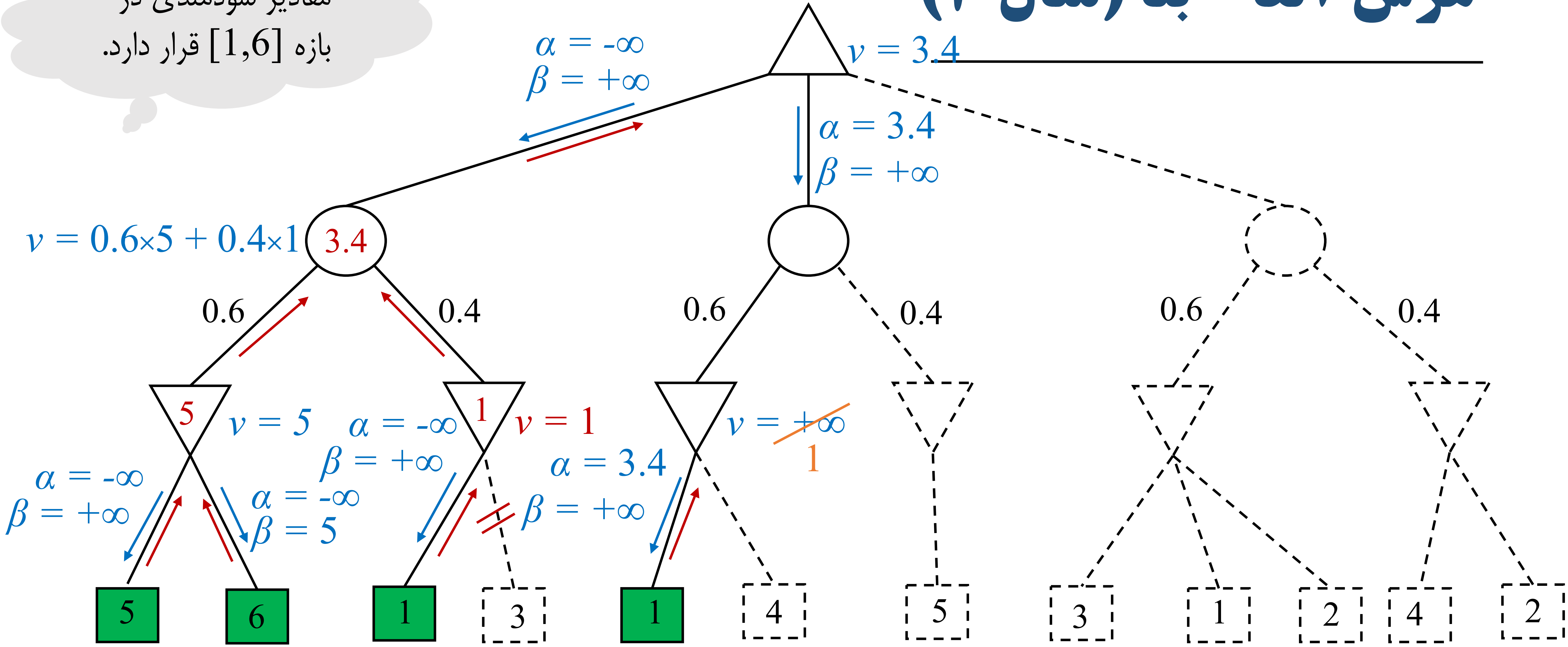
هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در
بازه [1,6] قرار دارد.



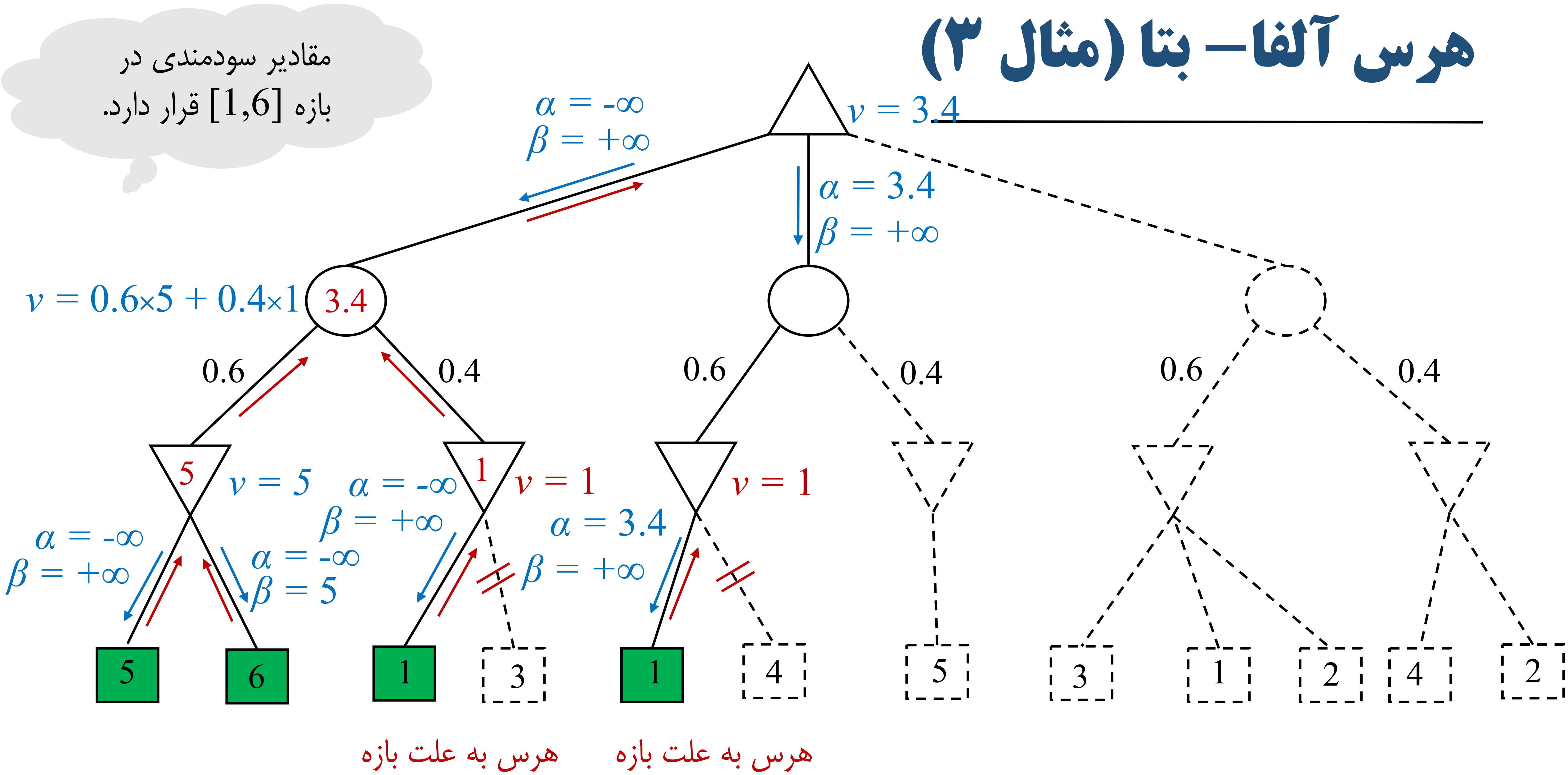
هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در
بازه [1,6] قرار دارد.

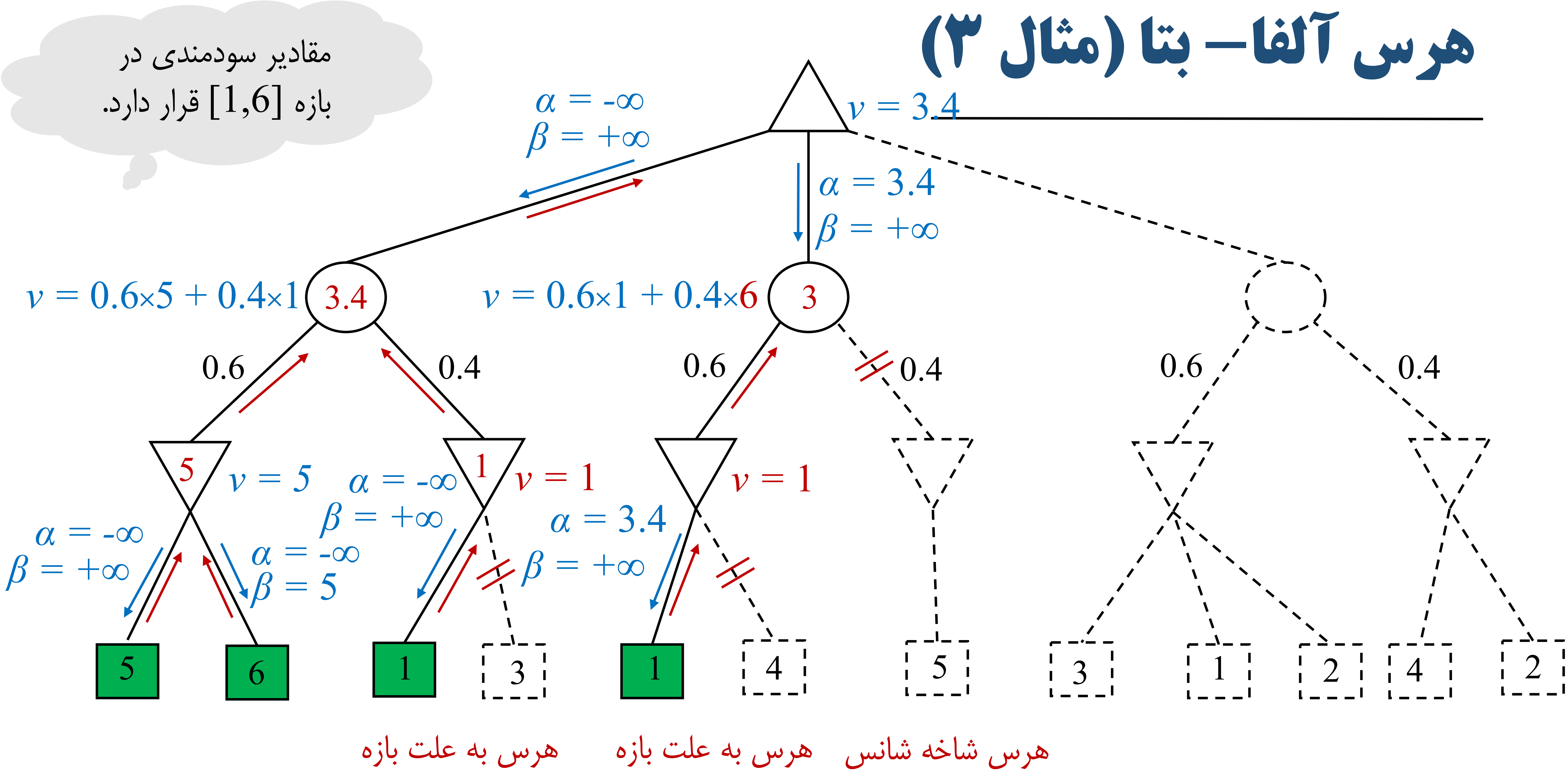


هرس به علت بازه

هرس آلفا– بتا (مثال ۳)



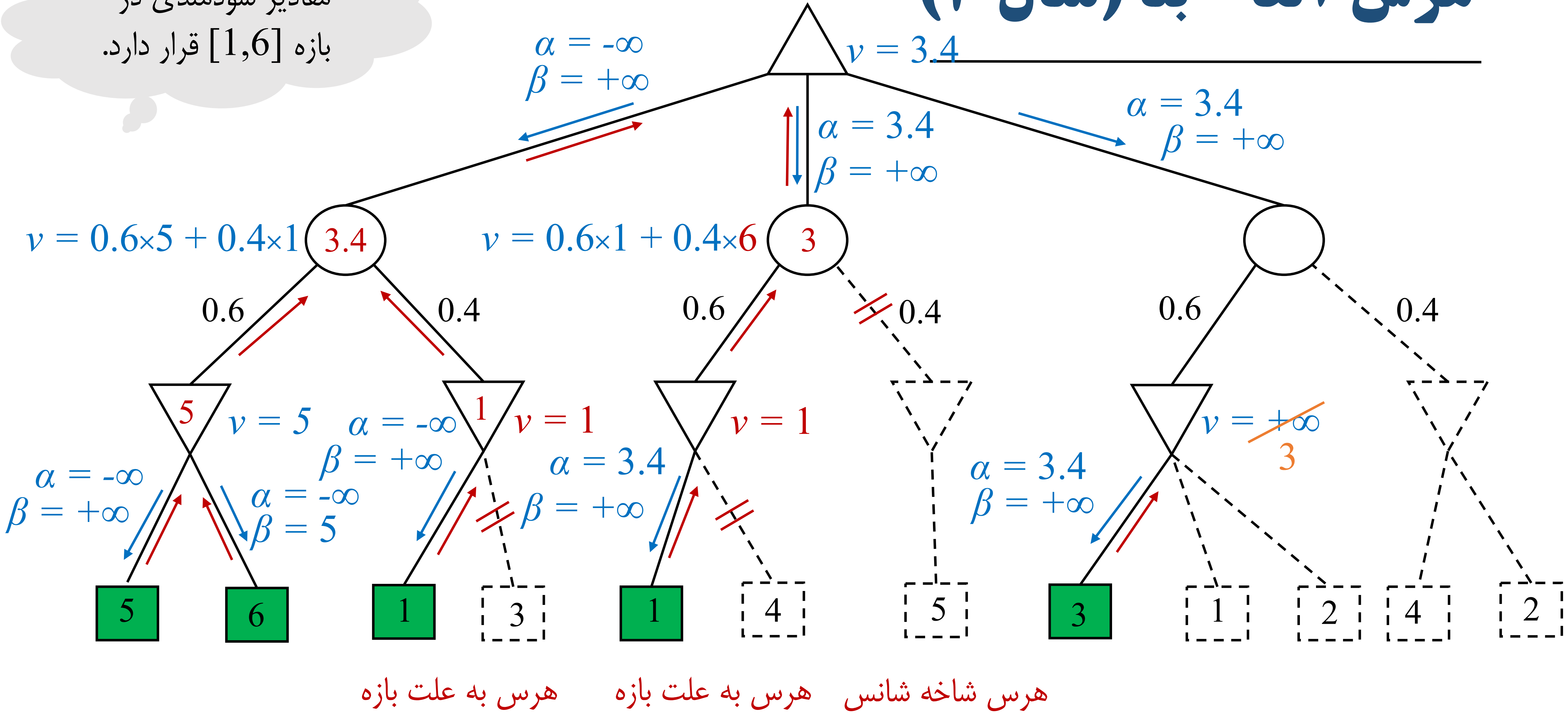
هرس آلفا– بتا (مثال ۳)



اگر شاخه سمت راست حداکثر سودمندی ممکن (۶) را داشته باشد، مقدار مورد انتظار گره شانس از حداقل سودمندی α کم‌تر است.

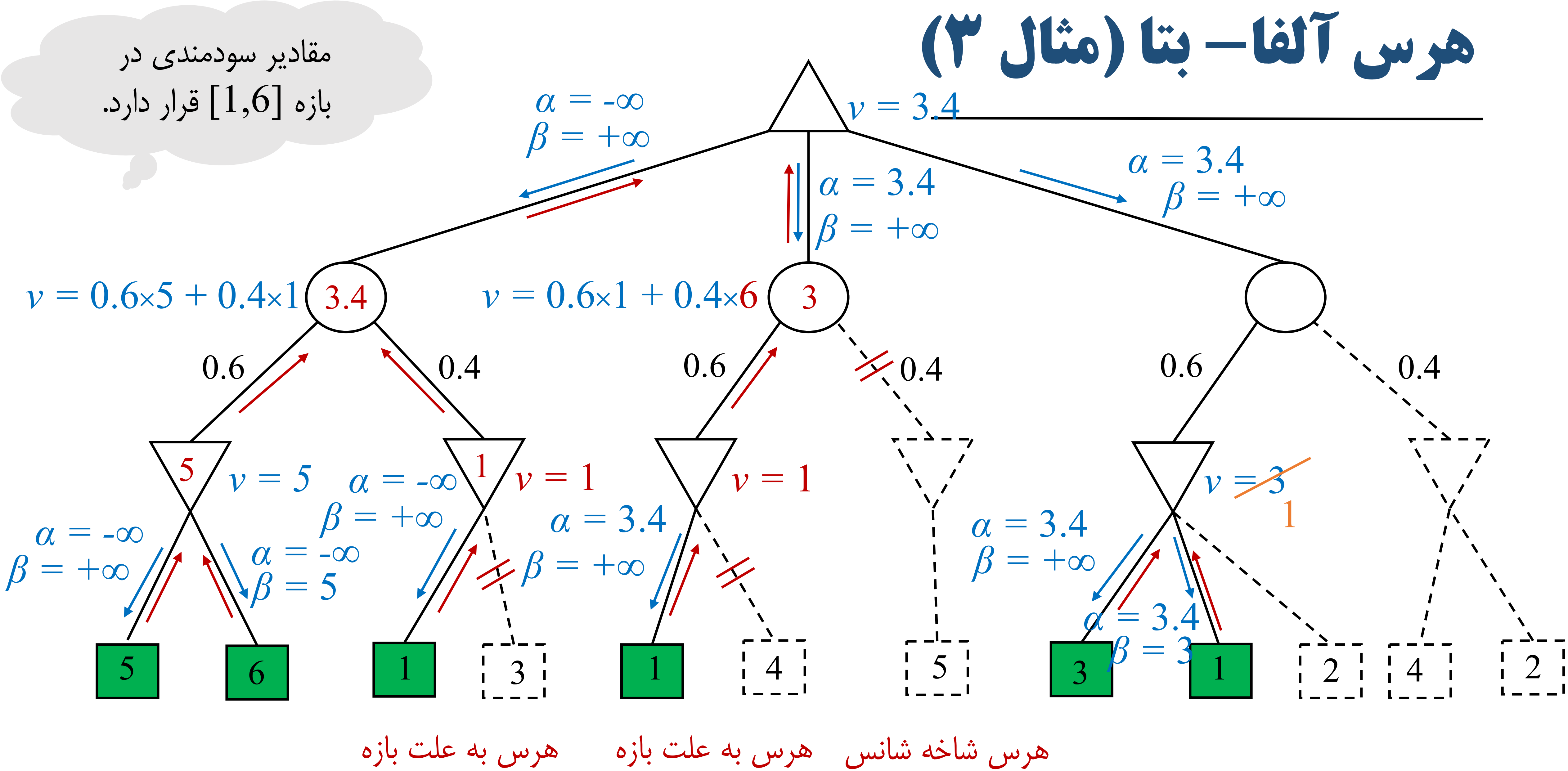
هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در
بازه [1,6] قرار دارد.



اگر شاخه سمت راست حداکثر سودمندی ممکن (۶) را داشته باشد، مقدار مورد انتظار گره شانس از حداقل سودمندی α کم‌تر است.

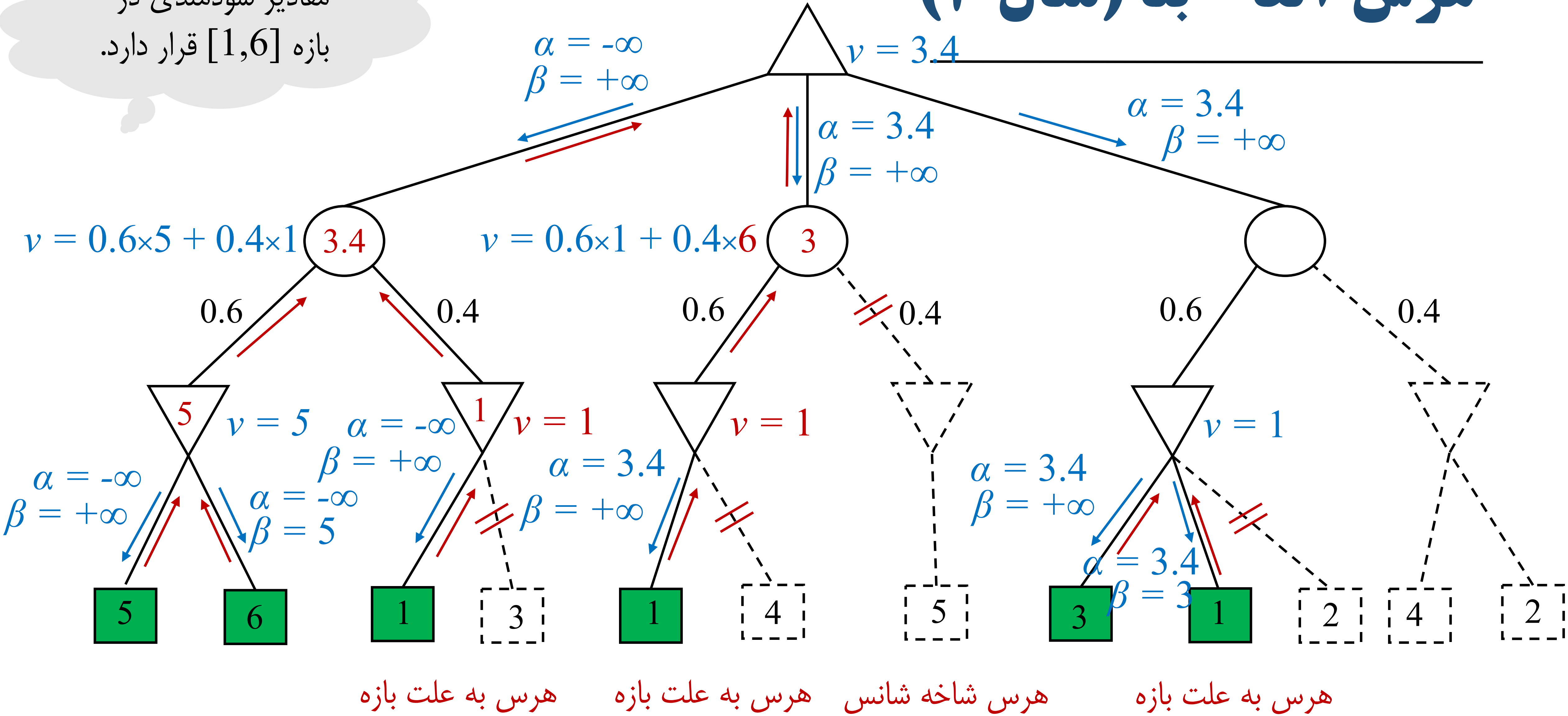
هرس آلفا–بتا (مثال ۳)



اگر شاخه سمت راست حداکثر سودمندی ممکن (۶) را داشته باشد، مقدار مورد انتظار گره شانس از حداقل سودمندی α کم‌تر است.

هرس آلفا– بتا (مثال ۳)

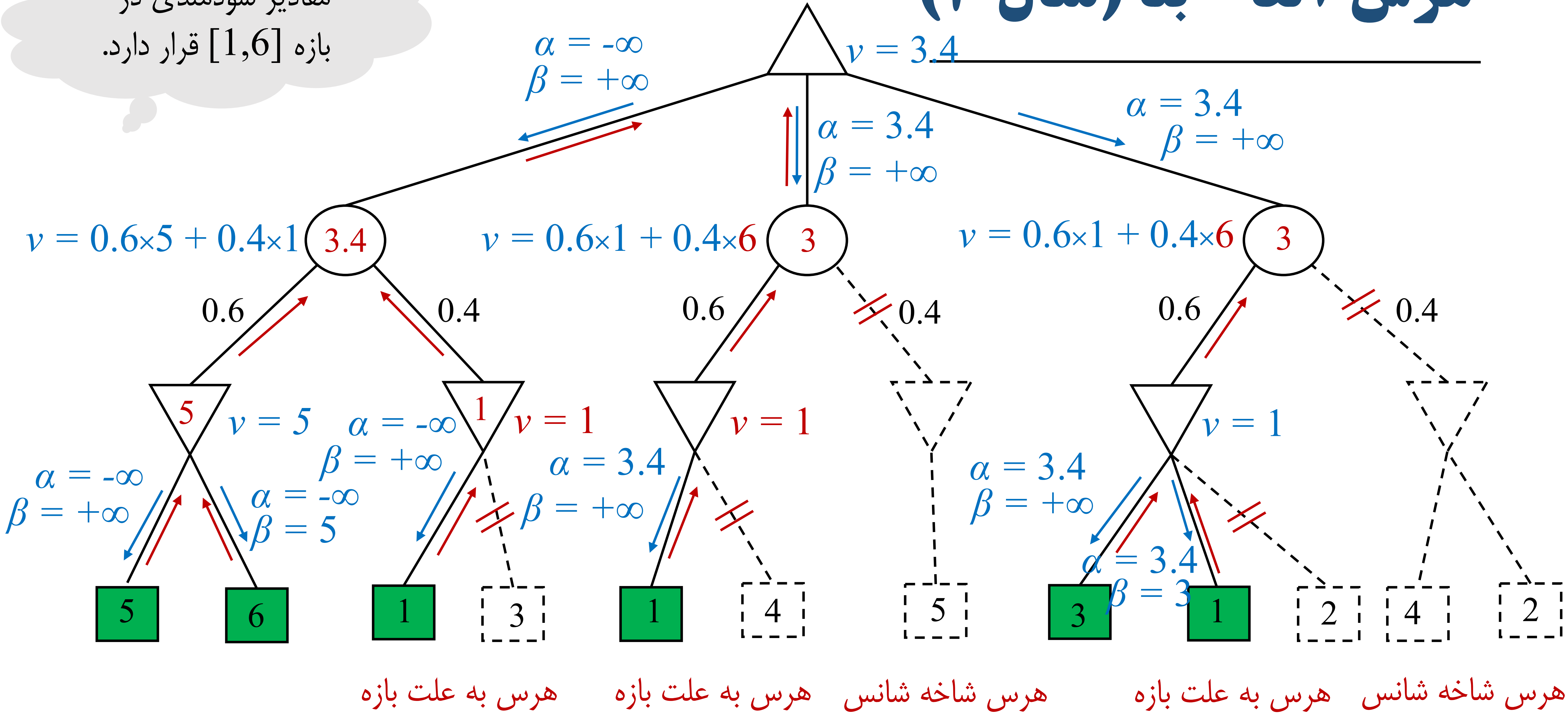
مقادیر سودمندی در
بازه [1,6] قرار دارد.



اگر شاخه سمت راست حداکثر سودمندی ممکن (۶) را داشته باشد، مقدار مورد انتظار گره شانس از حداقل سودمندی α کم‌تر است.

هرس آلفا– بتا (مثال ۳)

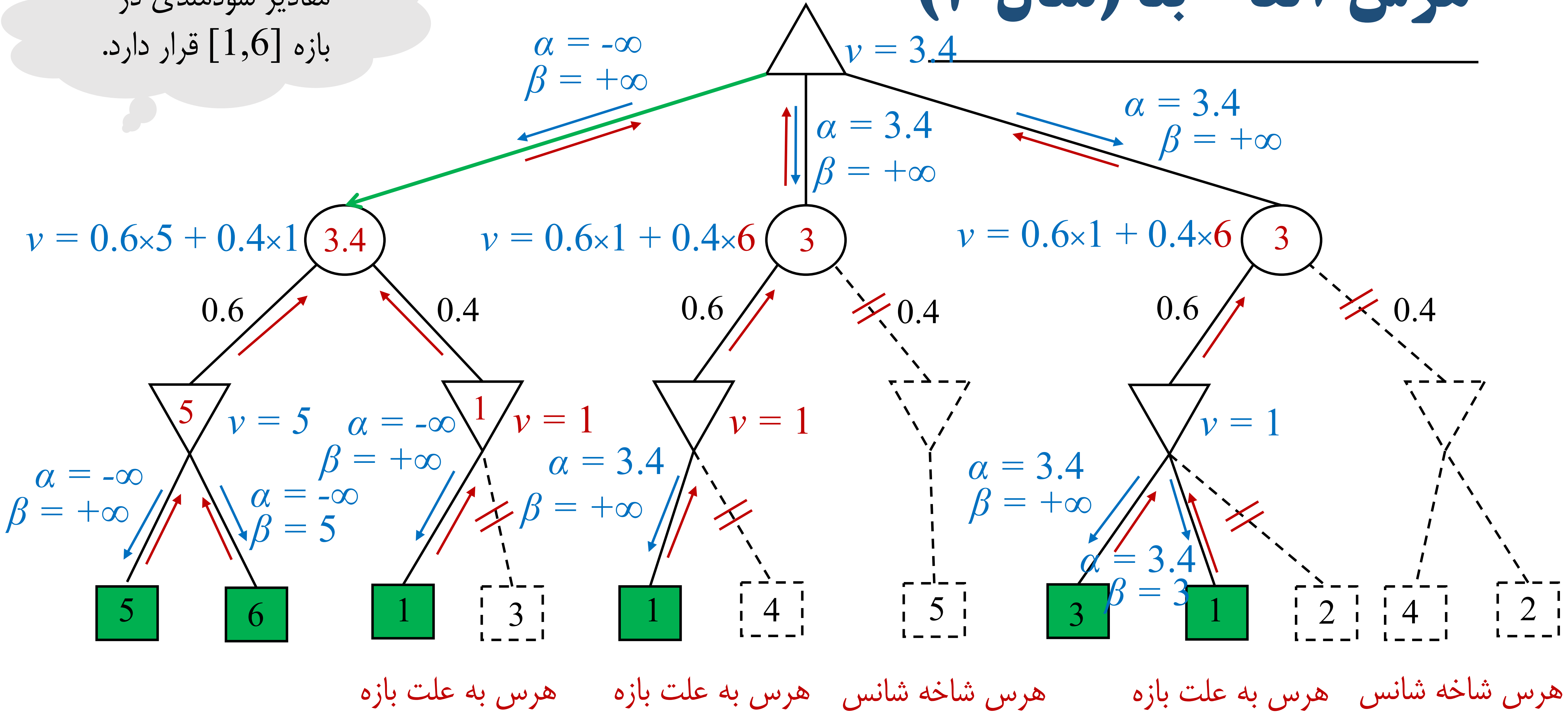
مقادیر سودمندی در
بازه [1,6] قرار دارد.



اگر شاخه سمت راست حداکثر سودمندی ممکن (۶) را داشته باشد، مقدار مورد انتظار گره شانس از حداقل سودمندی α کم‌تر است.

هرس آلفا– بتا (مثال ۳)

مقادیر سودمندی در بازه [1,6] قرار دارد.

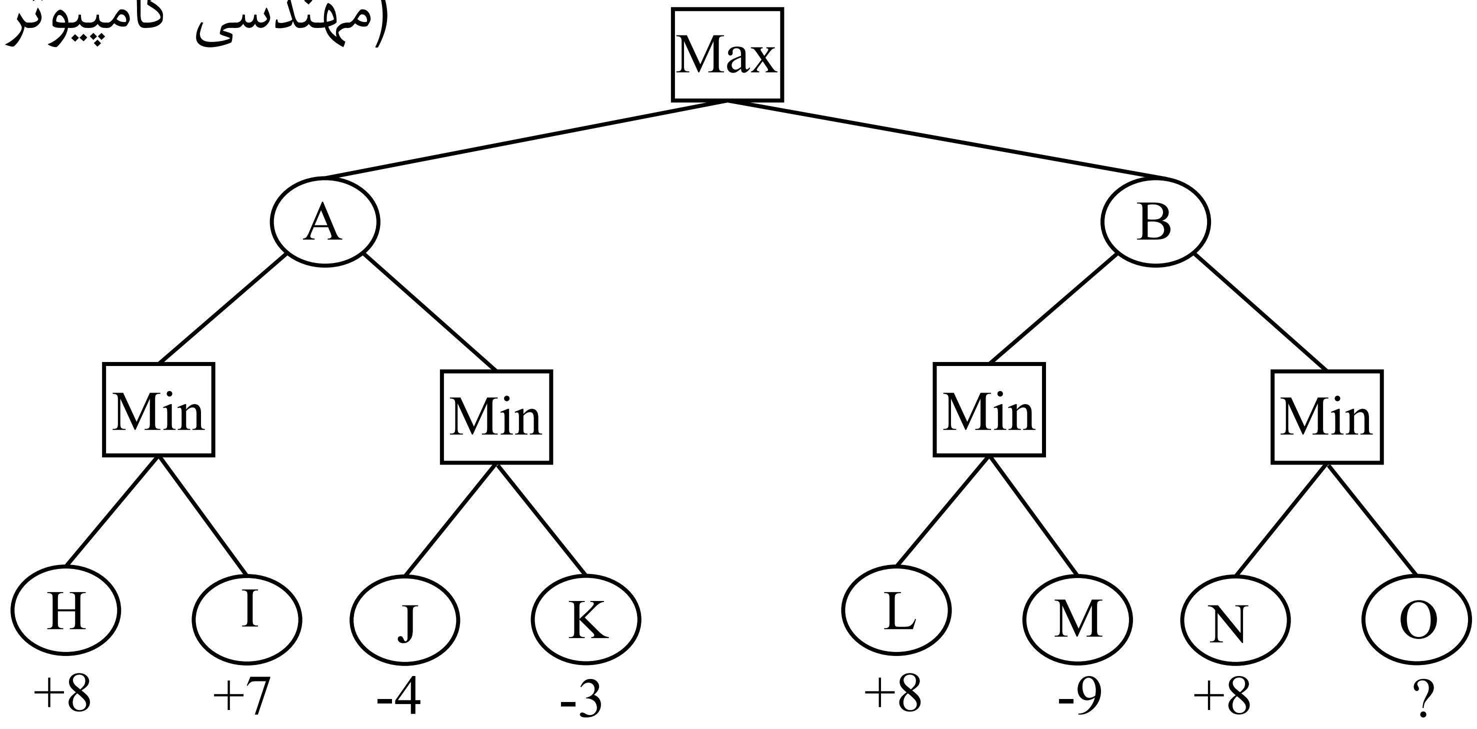


اگر شاخه سمت راست حداکثر سودمندی ممکن (۶) را داشته باشد، مقدار مورد انتظار گره شانس از حداقل سودمندی α کم‌تر است.

تست ۲

در درخت زیر، دو گره A و B از نوع گره شانس و معادل عمل تصادفی انداختن یک سکه هستند. درصورتی که بدانیم مقدار کمینه و بیشینه تابع ارزیابی گره‌های برگ، به‌ترتیب برابر با ۱۰− و ۱۰+ است. در روش هرس آلفا–بتا کدام یک از گره‌های این درخت هرس خواهند شد؟

(مهندسی کامپیوتر دولتی ۹۳)



۱) O

۲) O، N ✓

۳) O، N، M

۴) O، N، K