

Hevristično preiskovanje

prof. dr. Marko Robnik Šikonja
December 2015

Reševanje problemov

- ✱ preiskovanje je osnovni mehanizem za reševanje problemov
- ✱ na mnogo algoritmov lahko gledamo kot na metode preiskovanja
- ✱ smiselno je poznati osnovne strategije preiskovanja

Predstavitev s prostorom stanj

- ✱ prostor stanj (graf, drevo)
- ✱ množica stanj (vsa dosegljiva stanja problema)

$$\mathbf{S} = \{S; S_Z \xrightarrow{*} S\}$$

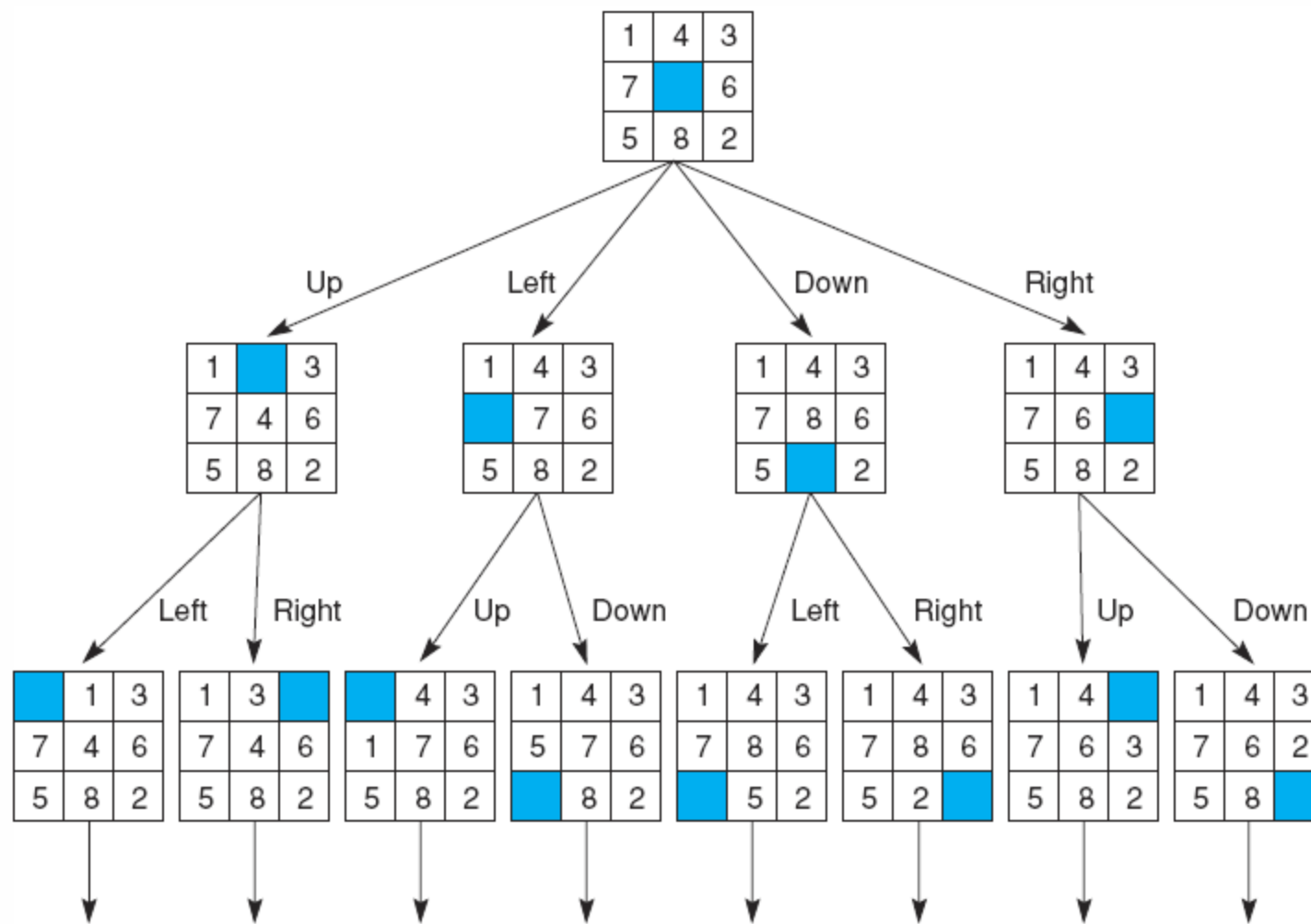
- ✱ povezave med stanji (prehodi med stanji, potrebujemo generator naslednikov)
- ✱ prostor stanj omogoča tudi enoten pogled na različne metode načrtovanja algoritmov

Prostor stanj

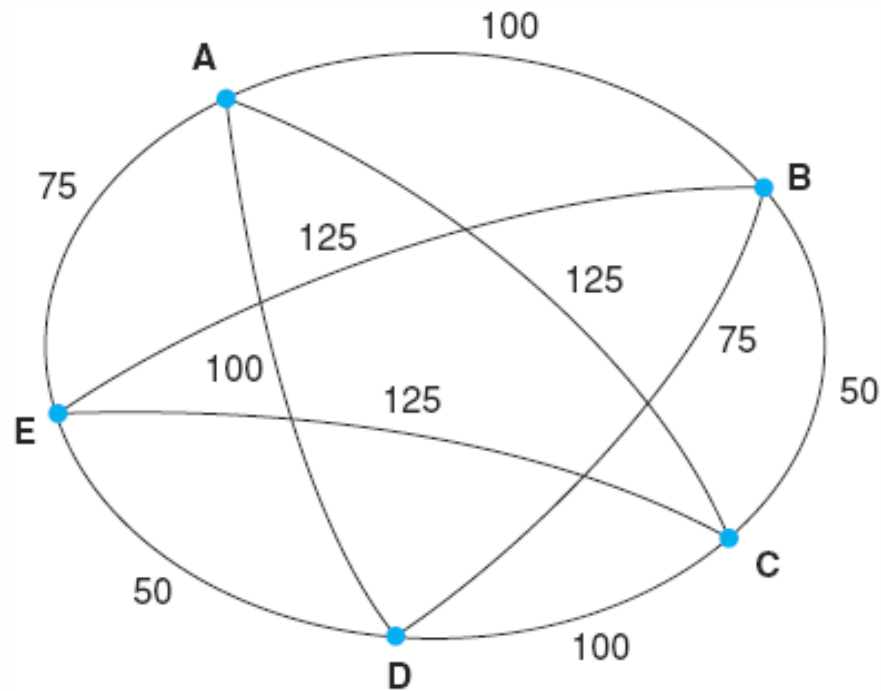
- ✱ Prostor stanj: $\mathbf{S} = \{S; S_Z \xrightarrow{*} S\}$
- ✱ kvaliteta stanja: $q(S)$
- ✱ končno stanje: $S_0 = \arg \max_{S \in \mathbf{S}} q(S)$
- ✱ različne strategije preiskovanja

Primer: drseče ploščice

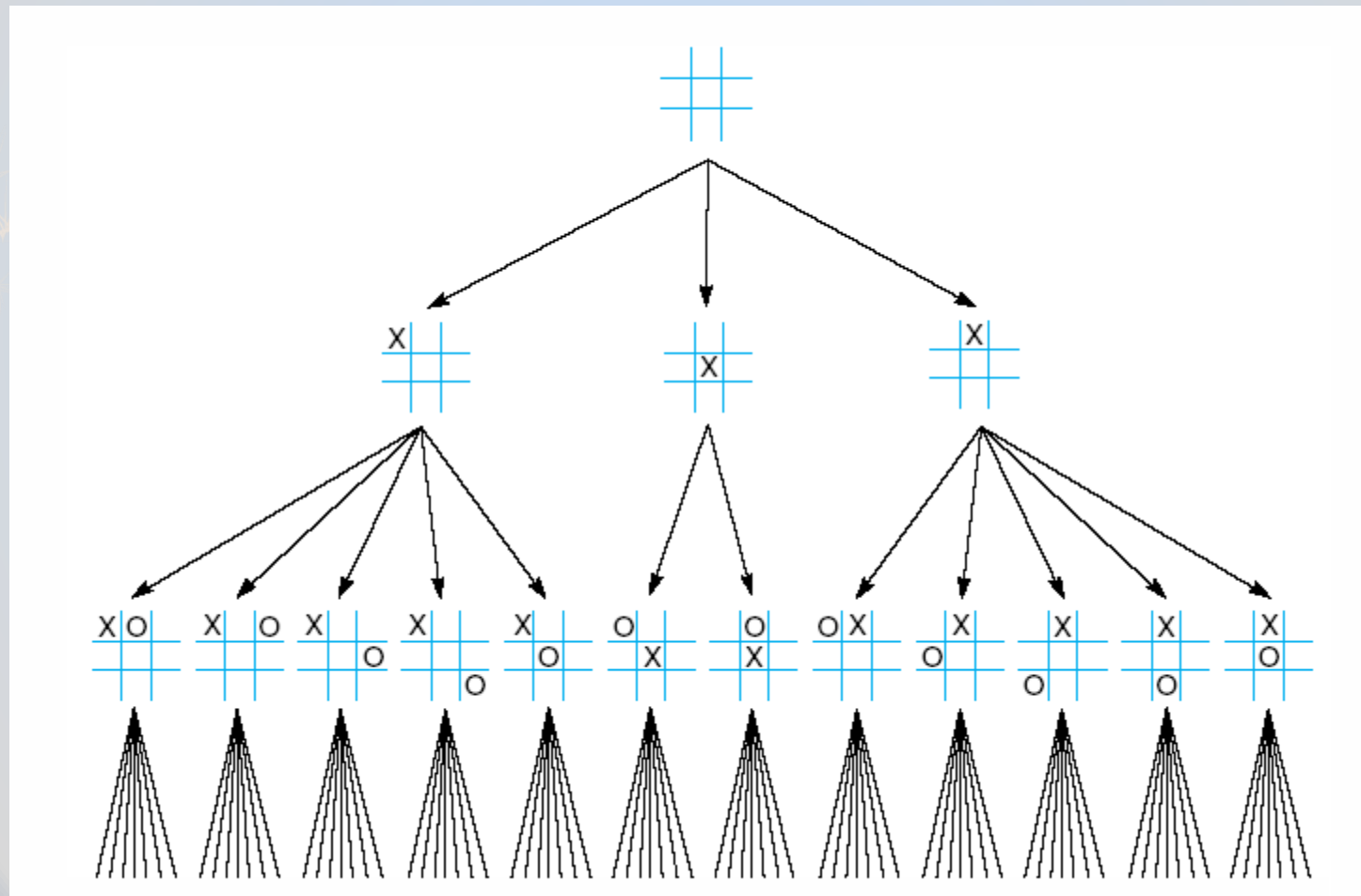
1	2	3
8		4
7	6	5



Primer: trgovski potnik



Primer: križci in krožci



Neinformirano preiskovanje

- ✱ v širino, v globino, iterativno poglobljanje

V globino iterativno

```
status depthFirst (start_state) {  
    stack.makemnull(); // inicializacija  
    stack.push(start_state); // začetno stanje  
    while ( ! stack.isEmpty() ) {  
        state = stack.pop (); // trenutno stanje je na vrhu  
        if (goal (state))  
            return SUCCESS;  
        else  
            stack.push( successors (state) ); // vsi nasledniki na vrh  
    }  
    return FAILURE ;  
}
```

V globino rekurzivno

```
status depth (state current ) {  
  if (goal (current))  
    return SUCCESS;  
  suc = successors (current);  
  foreach ( s in suc )  
    if ( depth(s) == SUCCESS )  
      return SUCCESS ;  
  return FAILURE ;  
}
```

V globino do določene globine rekurzivno

```
status depthRecursive (state current, int toDepth) {  
    if (goal (current))  
        return SUCCESS;  
    if (toDepth>0) {  
        suc = successors (current);  
        for each s in suc  
            if (depthRecursive(s, toDepth-1)==SUCCESS)  
                return SUCCESS  
    }  
    return FAILURE ;  
}
```

V širino

```
status breadthFirst (start_state) {  
    queue.makemnull(); // inicializacija  
    queue.enqueue(start_state); // začetno stanje v vrsto  
    while ( !queue.isEmpty() ) {  
        state = queue.dequeue(); // trenutno stanje je prvo v vrsti  
        if (goal (state))  
            return SUCCESS;  
        else  
            queue.enqueue(successors (state)); // na konec vrste  
        // closed.enqueue(state) ;  
    }  
    return FAILURE ;  
}
```

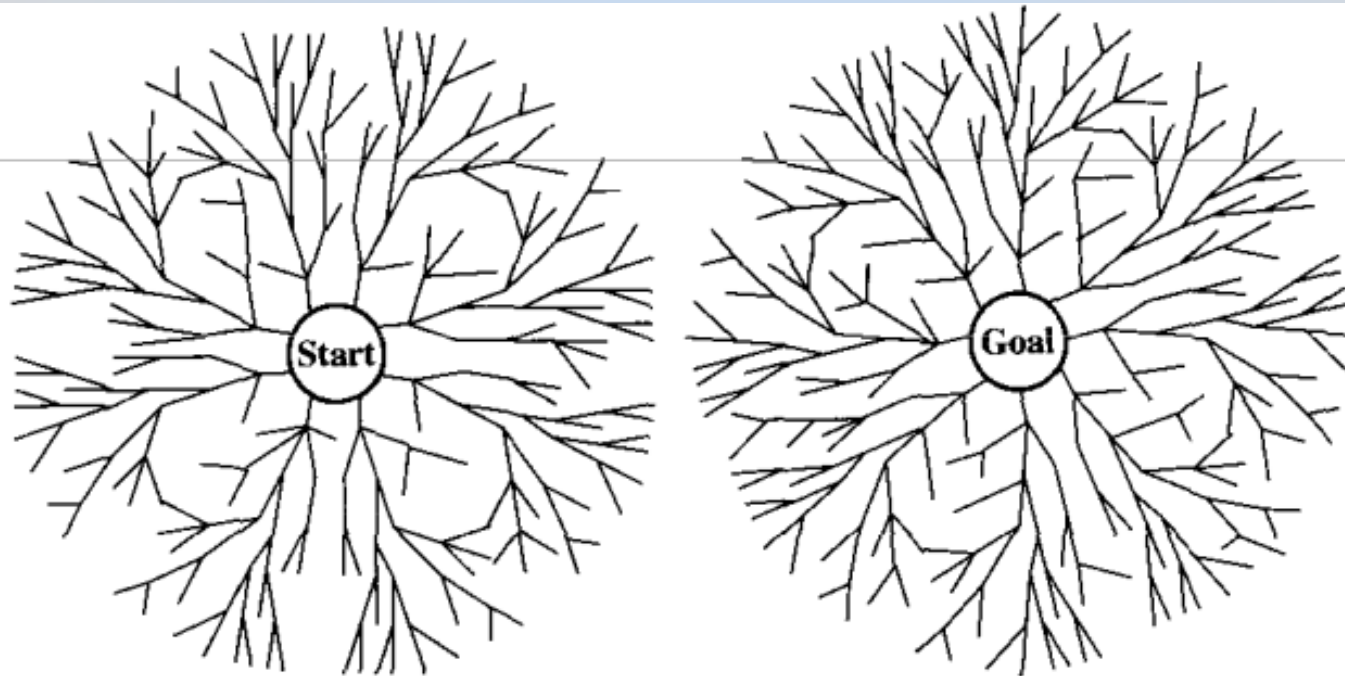
Iterativno poglabljanje

```
status ID (start_state) {  
    int depth = 1 ;  
    while (depth < MAX_DEPTH) {  
        state = depthRecursive(start_state, depth) ;  
        if (state == SUCCESS)  
            return SUCCESS ;  
        else depth++ ;  
    }  
    return FAILURE ;  
}
```

Zahtevnost

- ✱ odvisna od faktorja vejanja b in globine iskanja d
- ✱ število vozlišč pri iskanju
- ✱ v širino: $1 + b + b^2 + b^3 + \dots + b^d = (b^{d+1} - 1) / (b - 1) = O(b^d)$
- ✱ iterativno poglobljanje
 $d + 1 + db + (d-1)b^2 + (d-3)b^3 + \dots + b^d = O(b^d)$

Dvosmerno iskanje

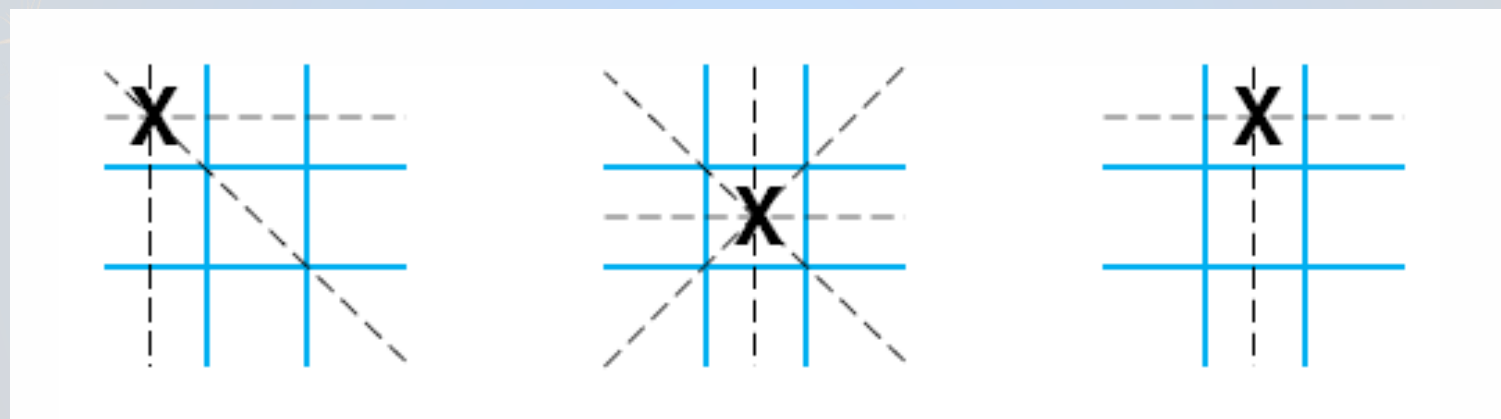


Hevristično preiskovanje

- ✿ hevrastika – metode in pravila za odkrivanje novega (Eureka!)
- ✿ uporaba:
 - ✧ točna rešitev ne obstaja
 - ✧ iskanje točne rešitve je računsko prezahtevno
- ✿ hevrastika: uporabna informacije, ki usmerja preiskovanje v obetavno smer

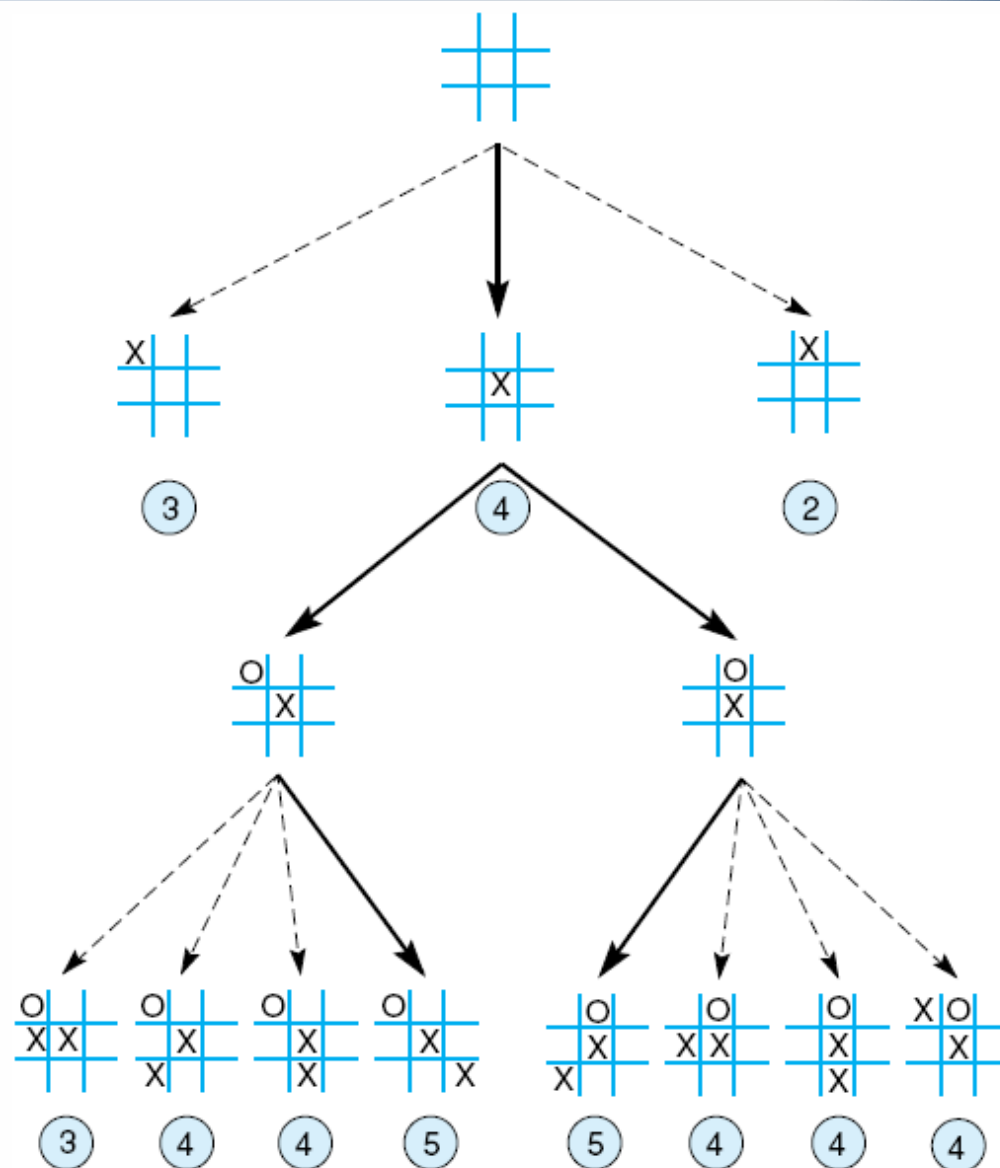
Primer: križci in krožci

- heuristika: na koliko načinov lahko zmagam



Križci in krožci

- simetrija
- heuristika



Požrešno preiskovanje

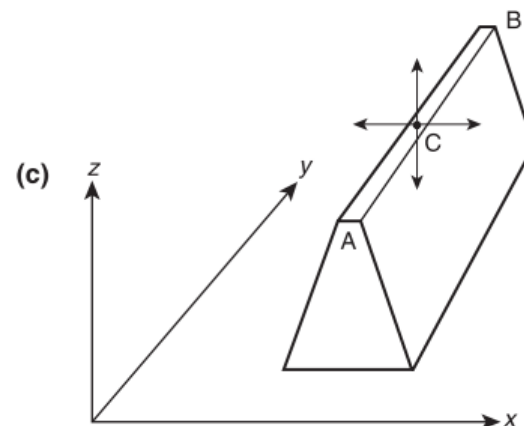
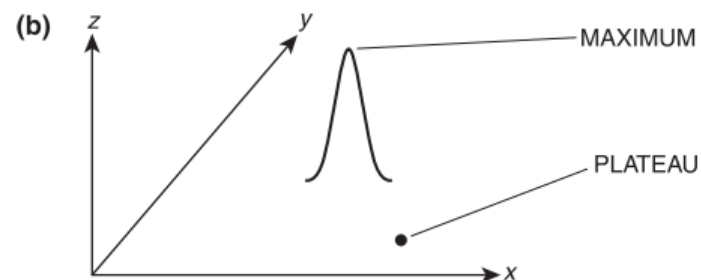
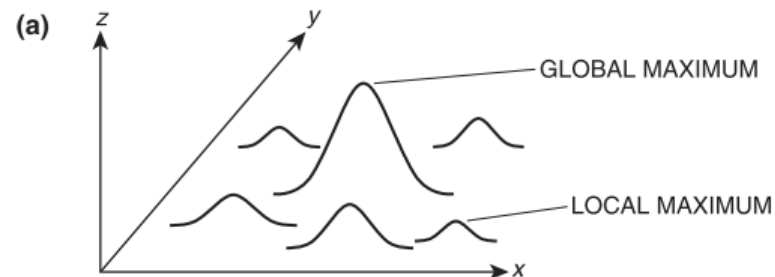
- ✱ hill-climbing
- ✱ na vsakem koraku izberemo najboljšega naslednika

Požrešno - implementacija

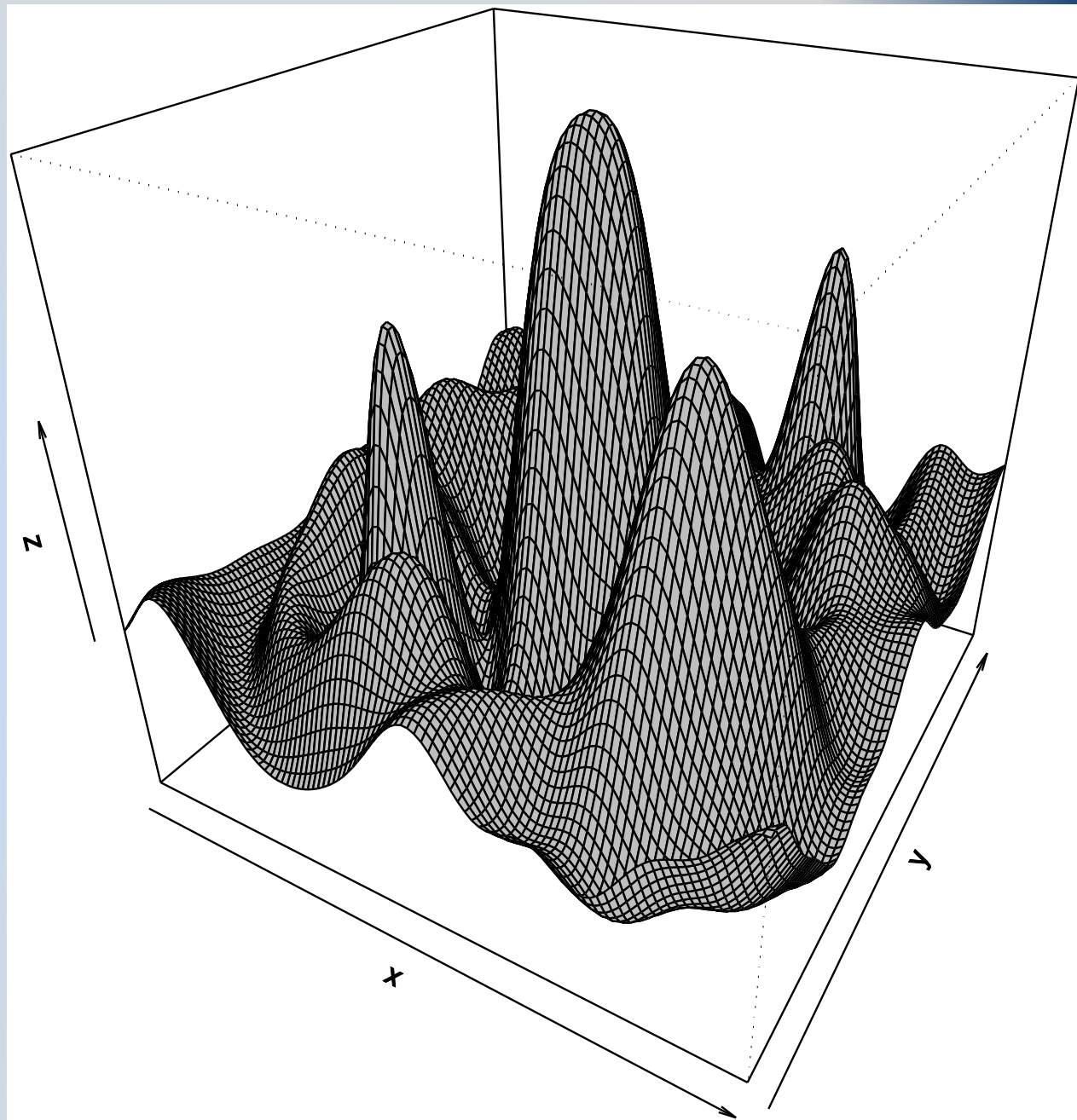
```
status hillClimbing (start_state) {  
    stack.makemull(); // inicializacija  
    stack.push(start_state); // začetno na vrh  
    while (!stack.isEmpty()) {  
        state = stack.pop (); // state = na vrhu sklada  
        if (goal (state))  
            return SUCCESS;  
        else {  
            suc = successors (state);  
            sort(suc); // uredi glede na hevristično oceno  
            stack.push(suc) ;  
        }  
    }  
    return FAILURE ;  
}
```

Lokalni ekstremi

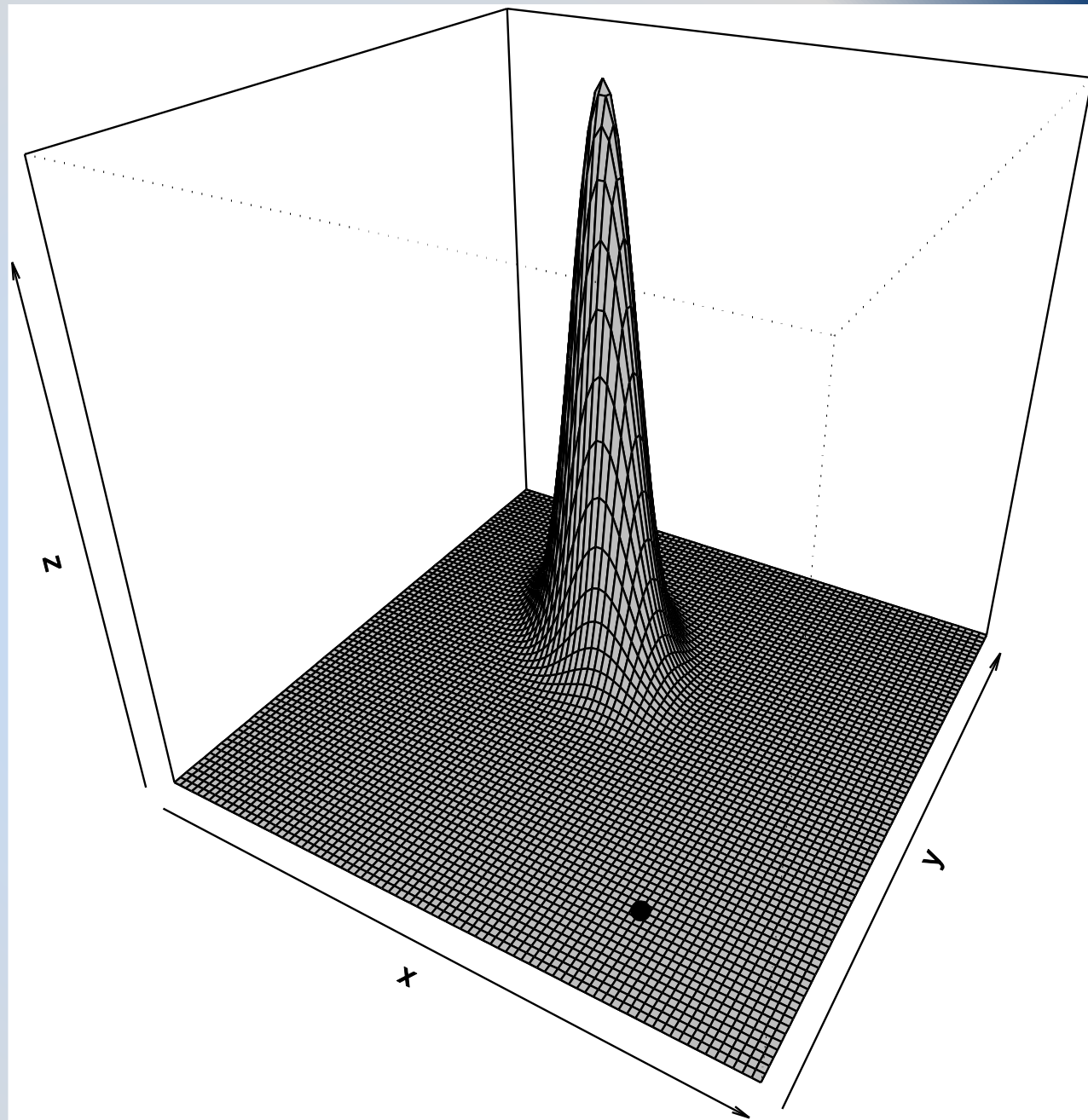
- lokalni in globalni maksimumi
- plato
- greben



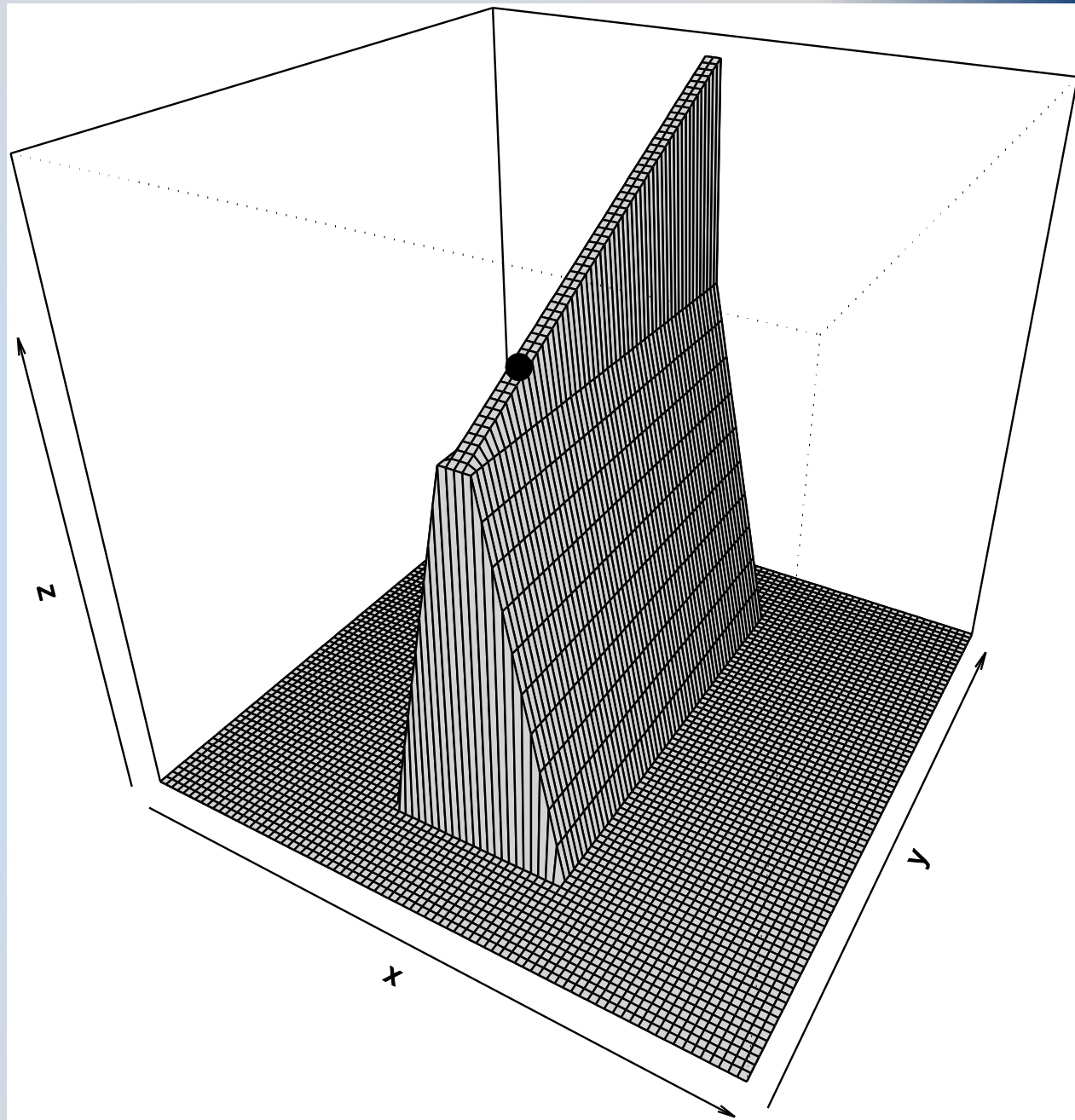
Lokalni ekstremi



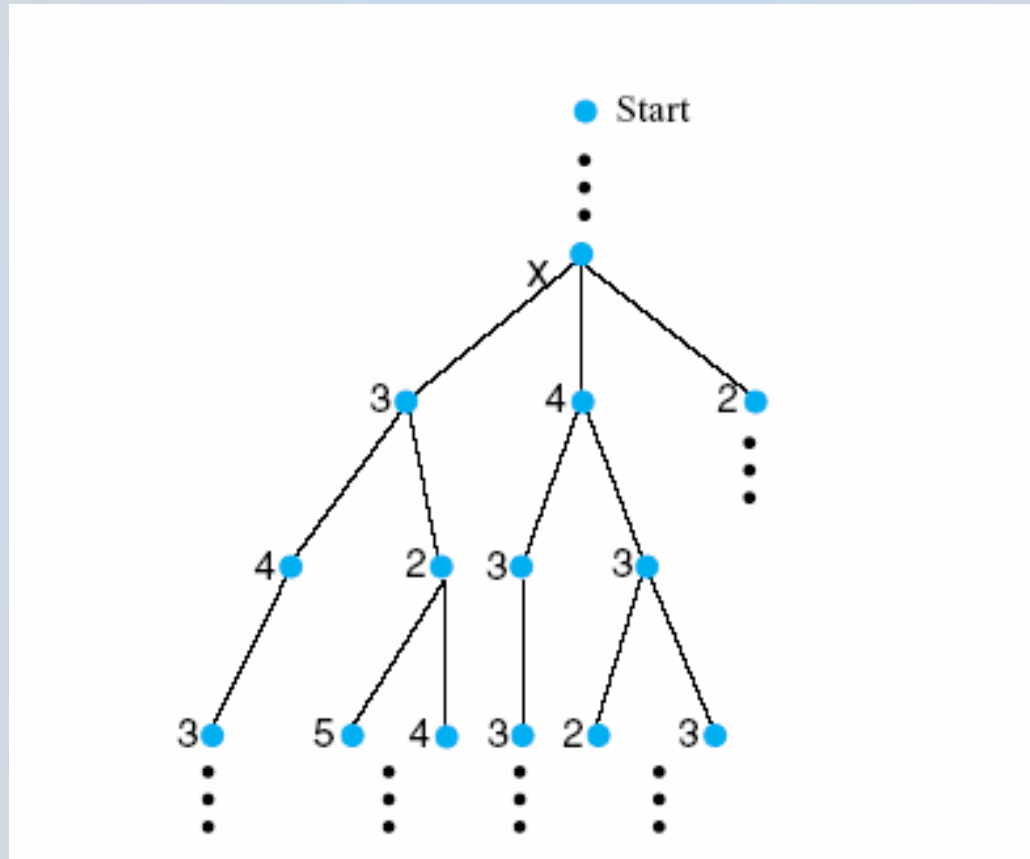
Plato



Greben



Pogled naprej (lookahead) - horizont



Najprej najboljši

- ✱ best first search
- ✱ vozlišča hranimo v prioritetni vrsti
- ✱ prioriteto določa hevristična ocena

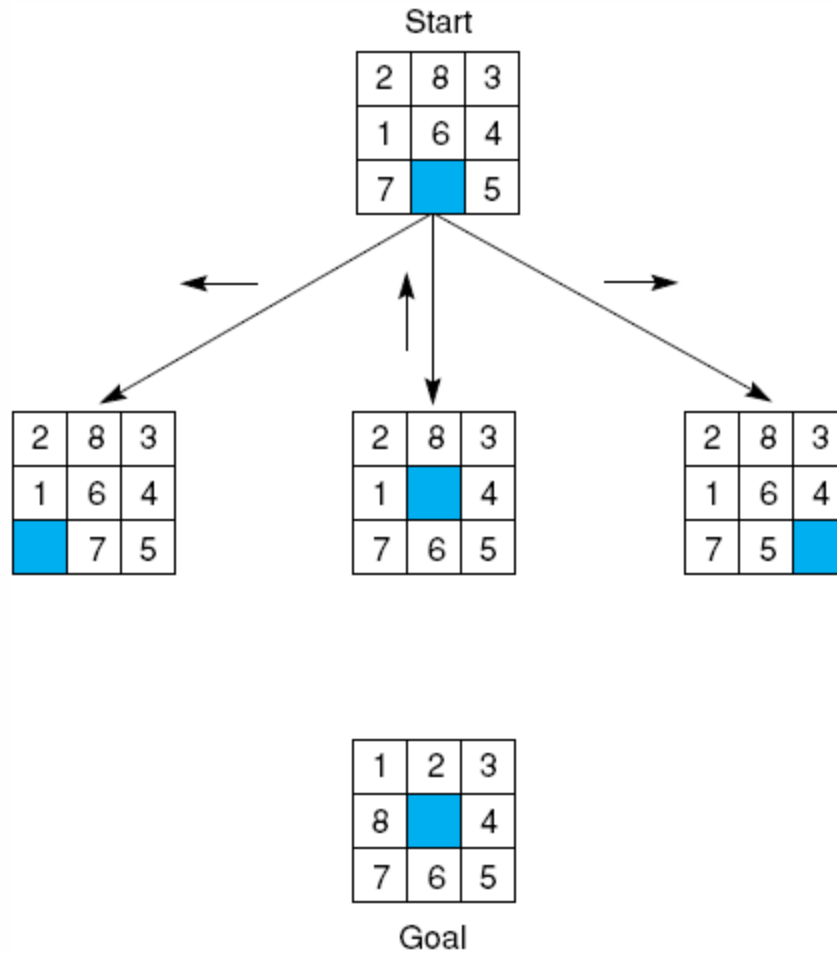
Najprej najboljši - implementacija

```
status bestFirst (start_state) {  
    priorityQueue.makeNull(); // inicializacija  
    priorityQueue.enqueue(start_state); // začetno stanje  
    while ( !priorityQueue.isEmpty() ) {  
        state = priorityQueue.deleteMin(); // prvi po prioriteti  
        if (goal (state))  
            return SUCCESS;  
        else  
            priorityQueue.addSorted(successors (state)); // urejeno dodajanje  
            // closed.enqueue(state) ;  
    }  
    return FAILURE ;  
}
```

A in A*

- ✱ poseben primer iskanja “najprej najboljši”
- ✱ $f(n) = g(n) + h(n)$
- ✱ n =stanje
- ✱ $g(n)$ = razdalja (vrednost, število korakov) od začetka do n
- ✱ $h(n)$ = ocena za pot od n do končnega stanja
- ✱ vozlišča preiskujemo urejeno glede na $f(n)$

Primer: drseče ploščice



Drseče ploščice: heuristike

1. ploščice, ki niso na mestu
(uporablja le del informacije)
2. vsota razdalj (premikov) do pravega mesta
(ne upošteva težavnosti zamenjav)
3. število direktnih zamenjav * 2

Hevristike: drseče ploščice

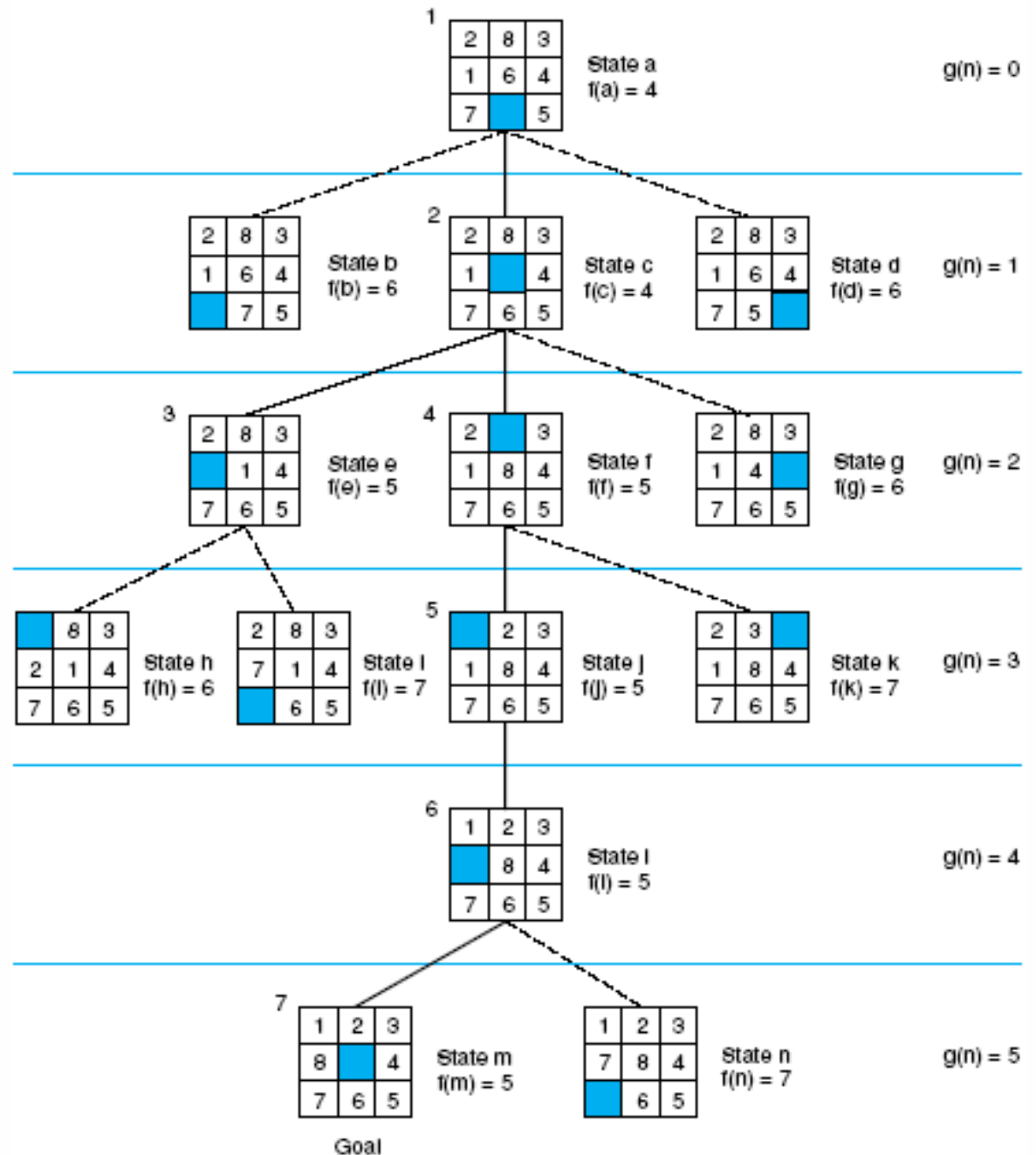
<table><tr><td>2</td><td>8</td><td>3</td></tr><tr><td>1</td><td>6</td><td>4</td></tr><tr><td></td><td>7</td><td>5</td></tr></table>	2	8	3	1	6	4		7	5	5	6	0
2	8	3										
1	6	4										
	7	5										
<table><tr><td>2</td><td>8</td><td>3</td></tr><tr><td>1</td><td></td><td>4</td></tr><tr><td>7</td><td>6</td><td>5</td></tr></table>	2	8	3	1		4	7	6	5	3	4	0
2	8	3										
1		4										
7	6	5										
<table><tr><td>2</td><td>8</td><td>3</td></tr><tr><td>1</td><td>6</td><td>4</td></tr><tr><td>7</td><td>5</td><td></td></tr></table>	2	8	3	1	6	4	7	5		5	6	0
2	8	3										
1	6	4										
7	5											
	Tiles out of place	Sum of distances out of place	2 x the number of direct tile reversals									

1	2	3
8		4
7	6	5

Goal

Preiskovanje s hevristikio
“ploščice, ki niso na mestu”

$g(n)$ – težnja k iskanju v širino



Dopustnost

- ✿ marsikdaj želimo zagotovljeno najboljšo (optimalno) rešitev
- ✿ algoritem je dopusten (admissable), če zanesljivo najde najkrajšo pot, če ta obstaja
- ✿ predpostavimo: $f^*(n) = g^*(n) + h^*(n)$
 - ✿ $g^*(n)$ = najkrajša pot do n
 - ✿ $h^*(n)$ = najkrajša pot od n do cilja
 - ✿ algoritem, ki bi uporabljal hevristiko $f^*(n)$ bi bil dopusten

Algoritma A in A*

- ✱ algoritem A: najprej najboljši s $f(n)$
- ✱ $g(n)$ je približek za $g^*(n)$,
- ✱ če je $g(n)$ monotona (kar pomeni, da se ne zmanjšuje), je dopustna je tudi $f(n) = g(n) + h^*(n)$
- ✱ če je $h(n)$ dopustna, kar pomeni, da podcenjuje razdaljo do cilja oz. $h(n) \leq h^*(n)$, je dopusten tudi A, ki uporablja $f(n) = g(n) + h(n)$ in tak algoritem imenujemo A*
- ✱ trivialna, a neuporabna dopustna hevrstika je $h(n)=0$ za vsa vozlišča, ki A* spremeni v iskanje v širino

Optimalnost in dopustnost: dokaz s protislovjem

- ✱ Algoritem A* uporabimo na grafu z dvema ciljnim vozliščema G_1 , G_2 . Cena poti do G_1 je f_1 , cena poti do G_2 je f_2 , velja $f_2 > f_1$. Denimo, da je algoritem našel pot G_2 pred G_1 in torej ni našel optimalne rešitve.
- ✱ Poglejmo vozlišče n , ki je na optimalni poti od začetka do G_1 . Takšno vozlišče mora obstajati in ker ga algoritem še ni razširil in ker je h dopustna heuristika velja $f_1 \geq f(n)$. Po predpostavki, bi algoritem preiskal G_2 pred n , kar lahko stori le v primeru, če $f(n) \geq f(G_2)$. Izraza združimo in dobimo $f_1 \geq f(G_2)$. Ker je G_2 ciljno vozlišče velja $h(G_2) = 0$ in torej $f(G_2) = g(G_2)$. Tako dobimo $f_1 \geq g(G_2) = f_2$, kar je v nasprotju z začetno domnevo, da je G_2 bolj oddaljen kot G_1 .
- ✱ Dokaz potrjuje, da lahko A* najde le najcenejšo pot do cilja.

Monotonost (konsistentnost) hevristike

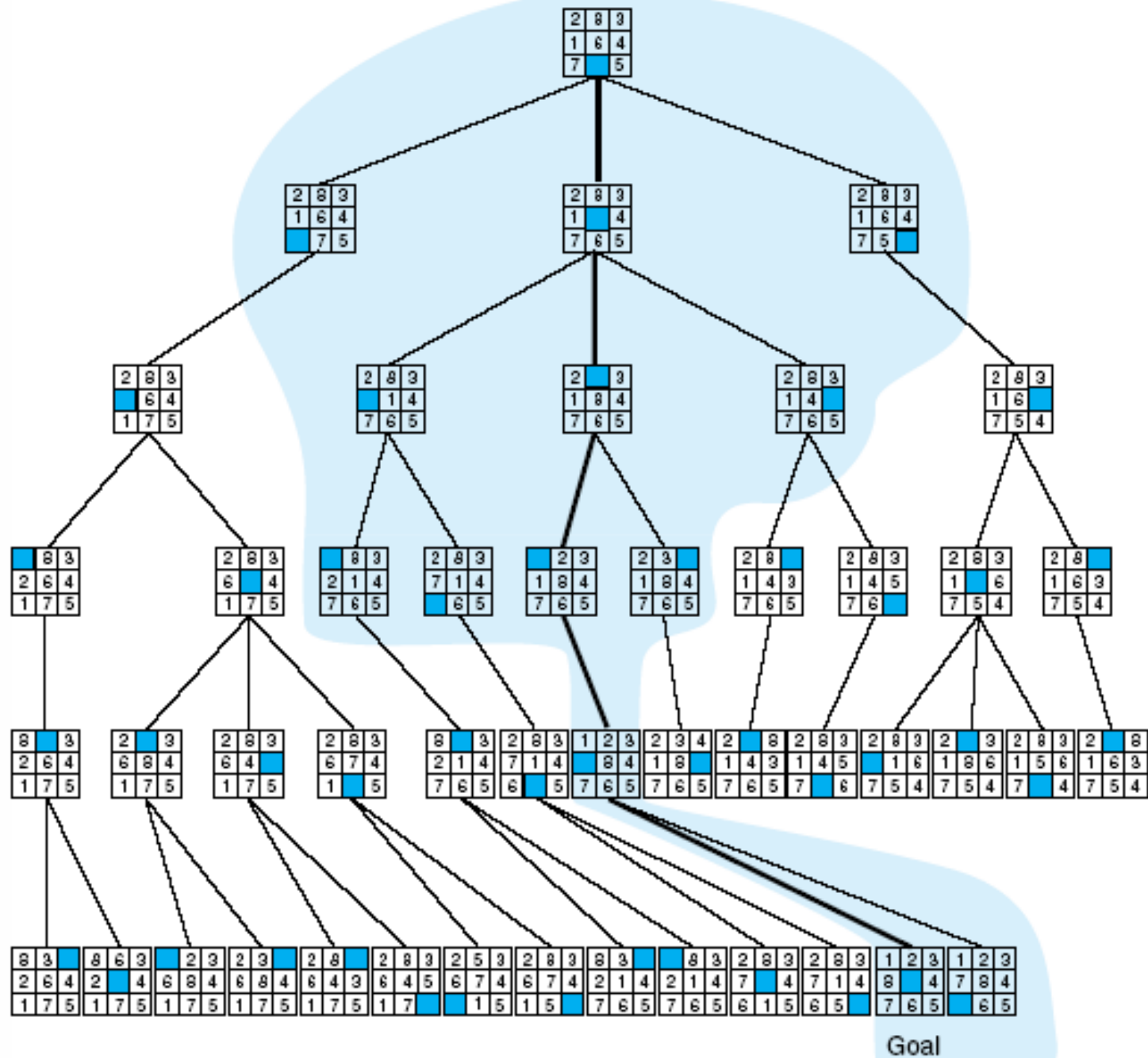
- ✿ ker ne zahtevamo $g(n)=g^*(n)$, lahko spočetka do nekaterih neciljnih vozlišč pridemo po daljši poti
- ✿ monotonost (lokalna dopustnost) pomeni, da do vsakega vozlišča pridemo po najkrajši poti, oziroma:
 - ✕ za vsa stanja n_i in n_j , kjer je n_j naslednik n_i
 $h(n_i) - h(n_j) \leq \text{cost}(n_i, n_j)$
kjer je $\text{cost}(n_i, n_j)$ dejanska razdalja med n_i in n_j
 - ✕ $h(\text{goal}) = 0$
- ✿ če je hevristika monotona, pri vsakem vozlišču že prvič vemo, da smo našli najkrajšo pot; ko zamenjamo oceno z dejansko vrednostjo, se $f(n)$ ne zmanjša in je monotono nepadajoča
- ✿ vsaka monotona hevristika je tudi dopustna

Informiranost heuristik

- ✱ želimo boljše heuristike
- ✱ za dve dopustni heuristiki h_1 in h_2 velja:
če za vsa stanja n velja: $h_1(n) \leq h_2(n)$ pravimo, da
je $h_2(n)$ boljše informirana
- ✱ presojamo kvaliteto in časovno zahtevnost
heuristike

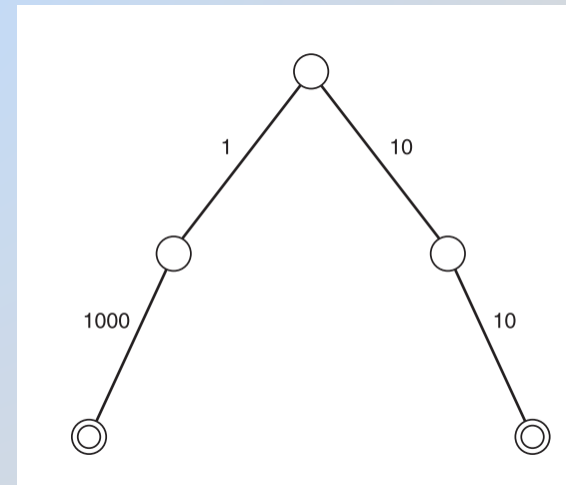
- $h_1(n)=0$

- $h_2(n)=$
“ploščice, ki
niso na
mestu”



Poenostavitve algoritma "najprej najboljši"

- ✱ $f(n) = g(n)$ oziroma $h(n)=0$
 - ✱ optimalno z monotonno naraščajočim $g(n)$
 - ✱ razveji in omeji (branch and bound),
 - ✱ podobno deluje tudi Dijkstra
- ✱ $f(n) = h(n)$
 - ✱ požrešno iskanje, neoptimalno



Razveji in omeji (branch and bound)

// vrača zgornjo mejo, do koder je še smiselno preiskovati
// kličemo z: best = null ; branchAndBound(start_state, ∞)
// n – trenutno stanje
// d – zgornja meja iskanja

```
int branchAndBound (state n, int d) {
```

```
    if ( g(n)+h(n)  $\geq$  d ) // omeji iskanje
```

```
        return  $\infty$  ;
```

```
    if ( goal( n ) ) {
```

```
        best = n ; // nova najboljša rešitev
```

```
        return g(n) ; // nova meja
```

```
    }
```

```
    suc = successors (n);
```

```
    foreach ( s  $\in$  suc ) // razveji
```

```
        d = min(d, branchAndBound(s, d) ) ; // išči in ažuriraj mejo
```

```
    return d ;
```

```
}
```


Izboljšave algoritma A*

- ✿ težava algoritma A* je velika poraba pomnilnika, saj je potrebno hraniti vsa vozlišča z vrednostjo $f(n)$ manjšo od ciljnega vozlišča
- ✿ izboljšave porabijo manj pomnilnika, a še vedno zagotavljajo optimalnost iskanja

Iterativno poglobljanje z A*

- ✱ Iterative-Deepening A* (IDA*)
- ✱ namesto poglobljanja globine iskanja, poglobljamo vrednost hevristične ocene $f(n)$
- ✱ deluje dobro pri problemih z malo različnih vrednosti $f(n)$ in zelo slabo pri npr. zveznih vrednostih $f(n)$

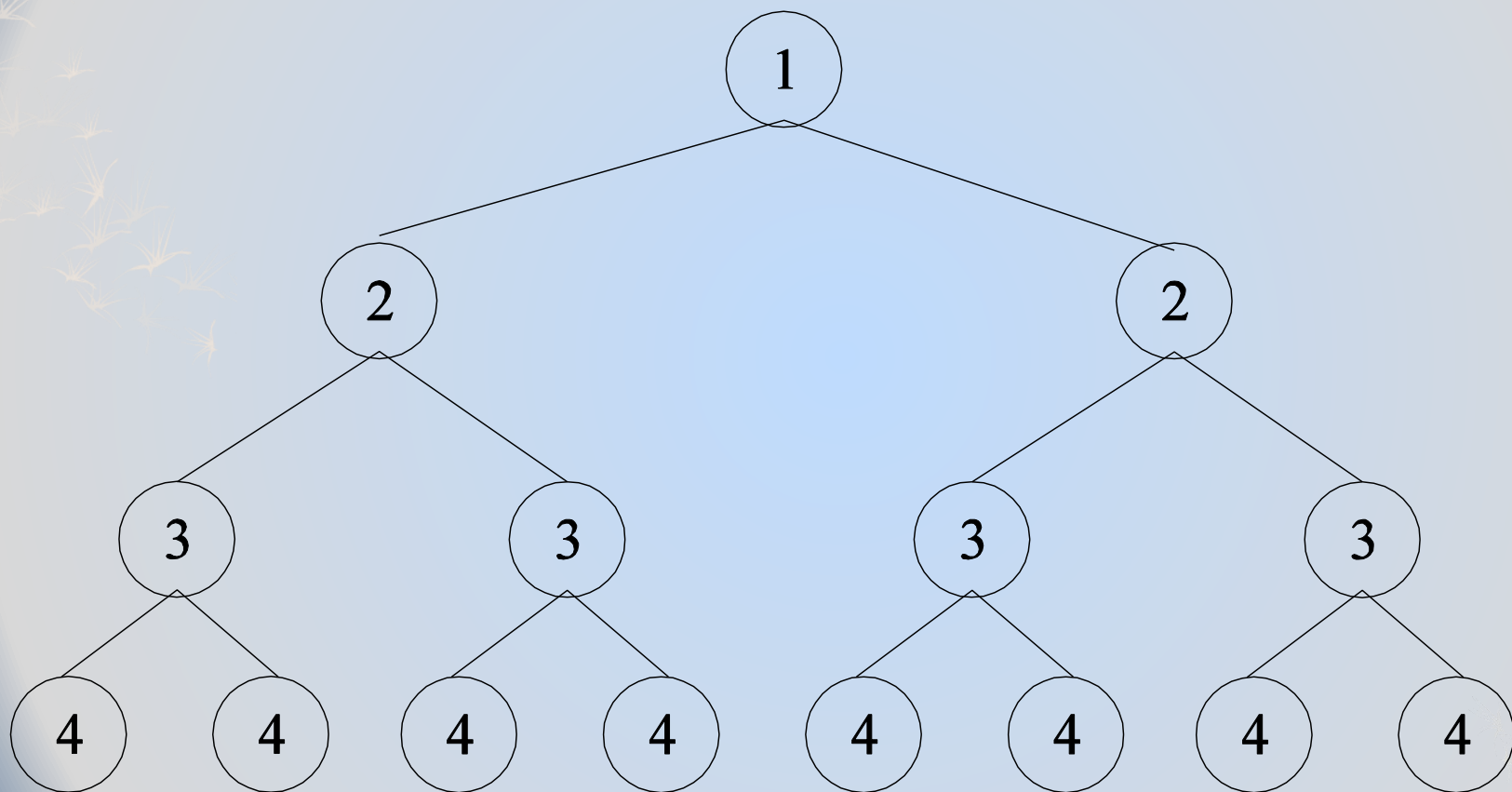
RBFS

- ✿ Recursive Best First Search, (Korf, 1993)
- ✿ shrani f vrednosti vseh otrok na trenutni poti,
- ✿ pri iskanju lahko zato preiskuje do meje sobratov
- ✿ pri vračanju si zapomni f vrednost najboljšega lista in zato ve, katere veje je vredno ponovno generirati
- ✿ prostorska zahtevnost $O(b \cdot d)$
- ✿ uspešnost odvisna od hevristične funkcije in potrebe po regeneriranju

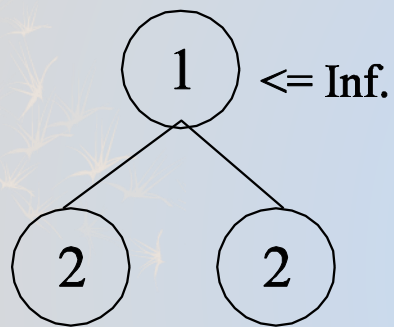
RBFS koda

```
double rbfs(searchNode node, int bound) {  
    if (node.f > bound)  
        return node.f ;  
  
    if ( goal(node) )  
        terminate_search(node) ;  
  
    children = getChildrenSorted(node) ;  
    if (children.length == 0)  
        return Infinite ;  
  
    while (children[0].f <= bound) {  
        if (children.length==1)  
            fSiebling = Infinite ;  
        else  
            fSiebling = children[1].f ;  
  
        children[0].f = rbfs(children[0], min(bound, fSiebling) ) ;  
        children.sort() ;  
    }  
    return children[0].f  
}
```

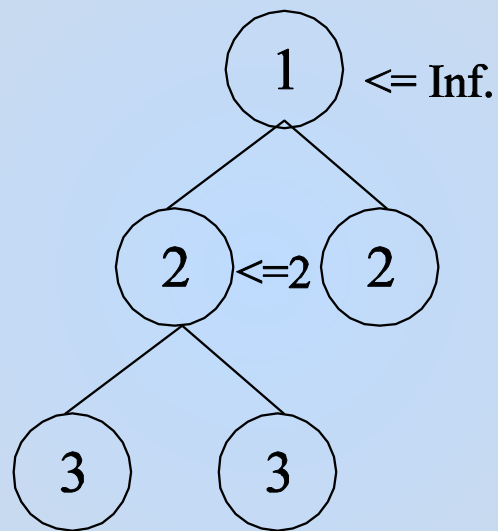
Prikaz delovanja RBFS: prostor stanj



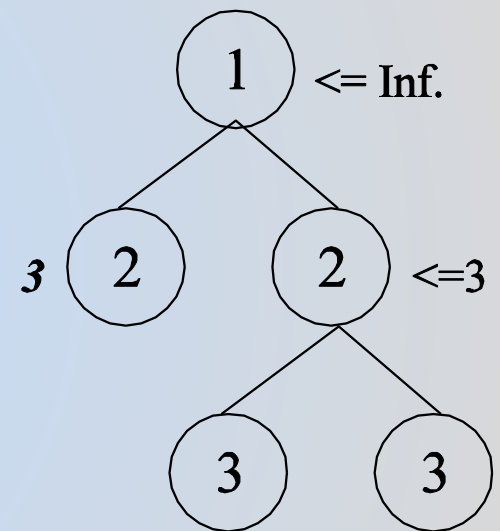
Prikaz delovanja RBFS



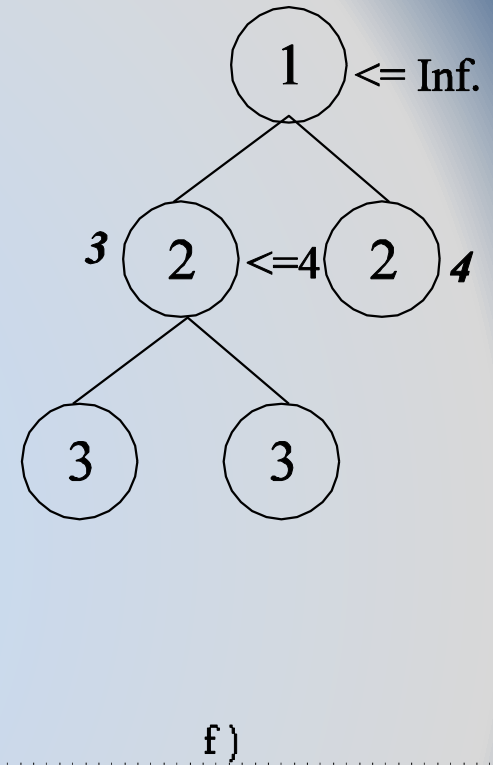
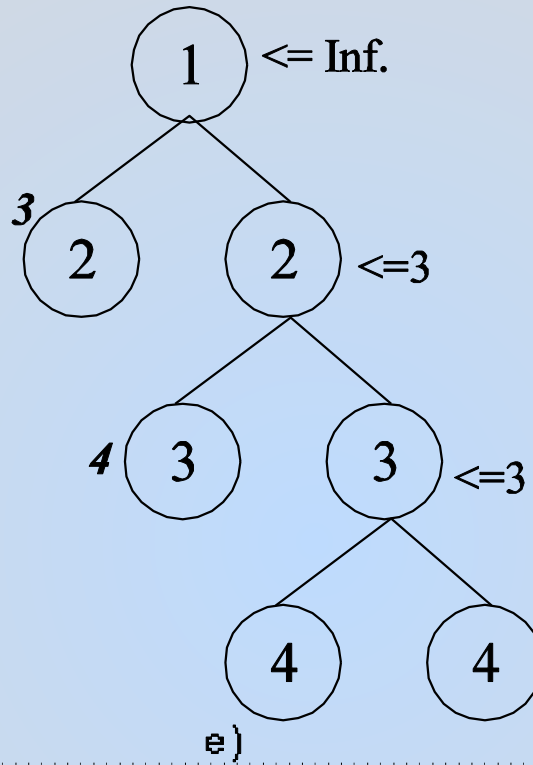
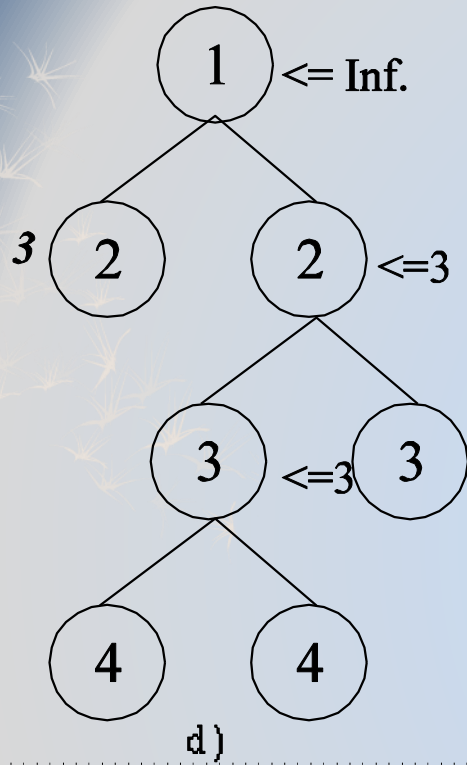
a)

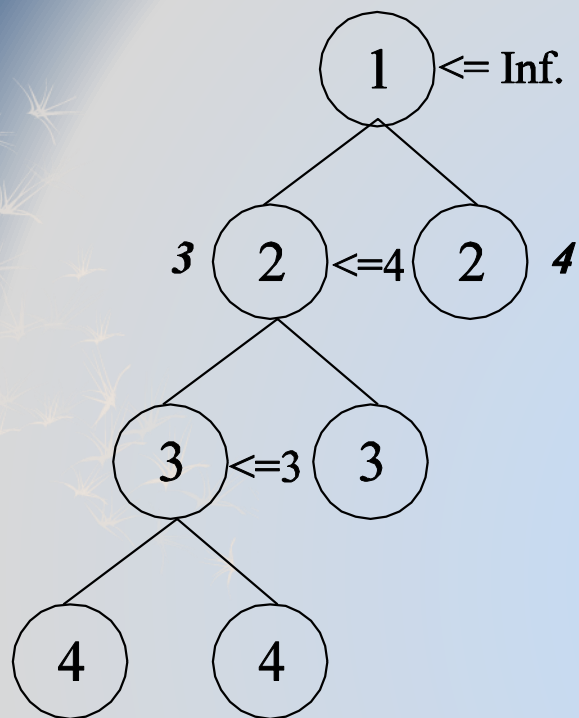


b)

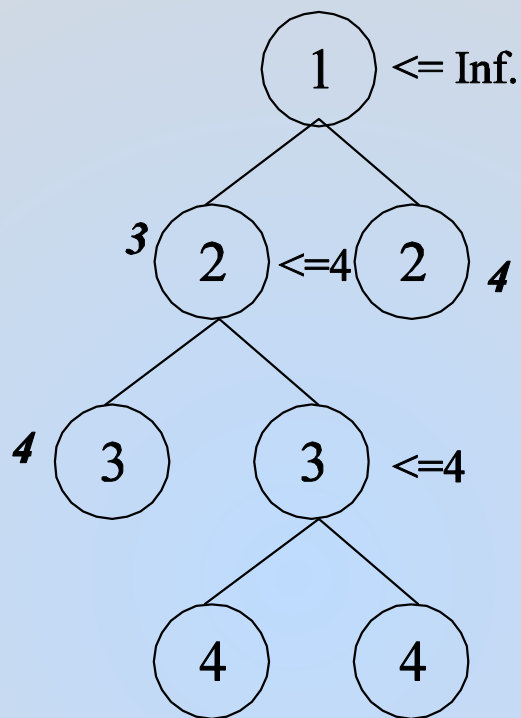


c)

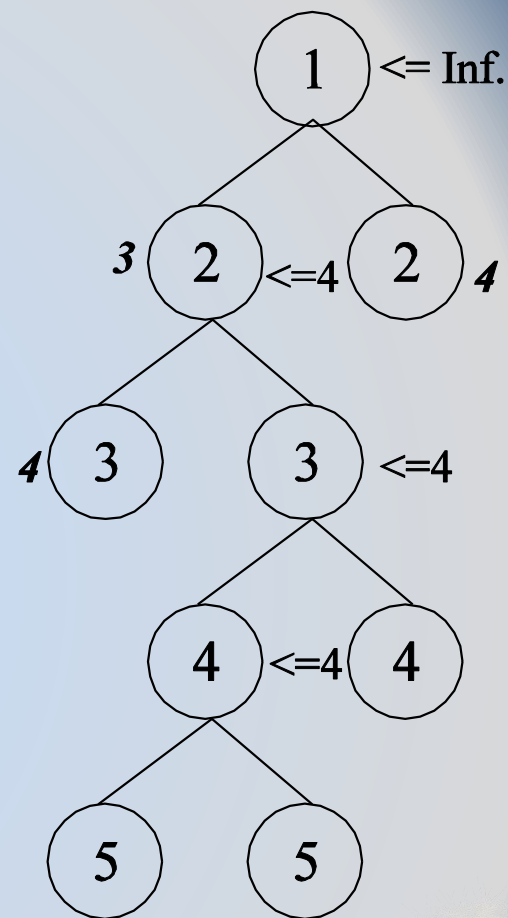




g)



h)



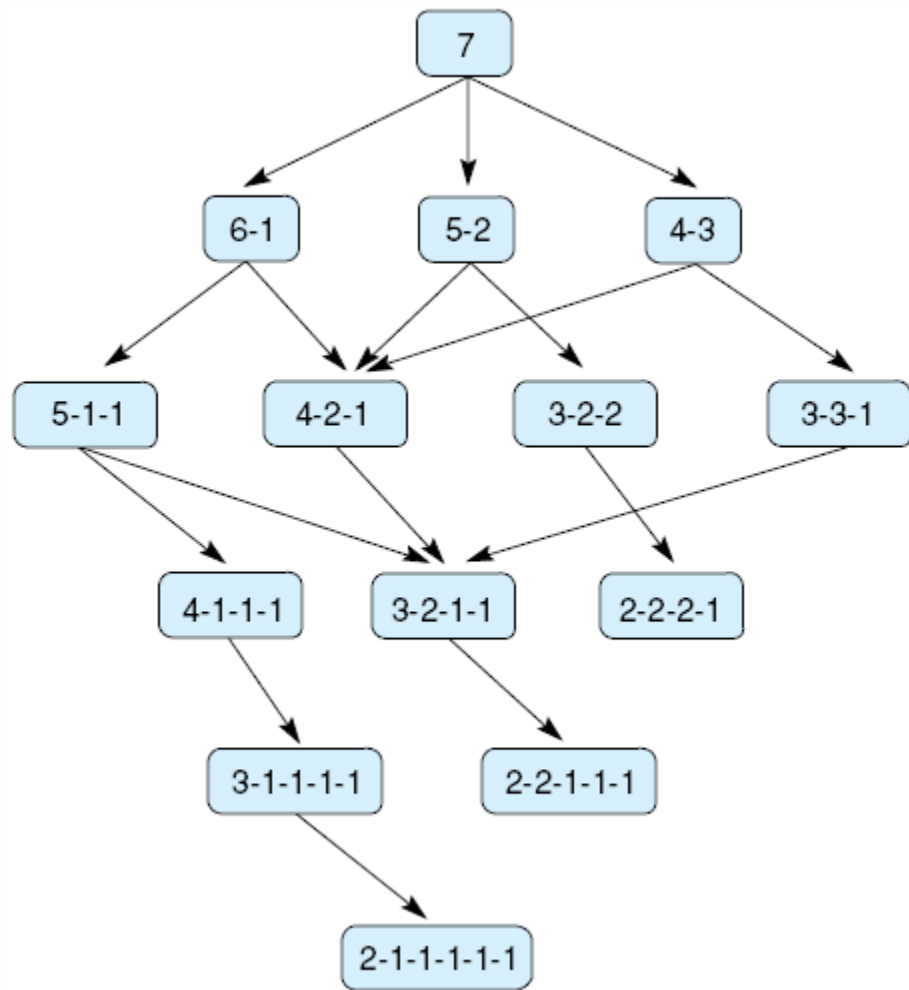
i)

MA* in SMA*

- ✿ pomnilniško omejena A* (memory bounded A*)
- ✿ deluje kot A*, dokler ne zmanjka pomnilnika, nato
 - ✗ odstrani list z najslabšo vrednostjo f
 - ✗ kot RBFS ažurira njegovo vrednost v predhodniku
 - ✗ če ima več listov enako vrednost f , odstrani najstarejšega, razširi najmlajšega
- ✿ SMA* najde optimalno rešitev kot A*

Hevristično preiskovanje in princip minimaksa

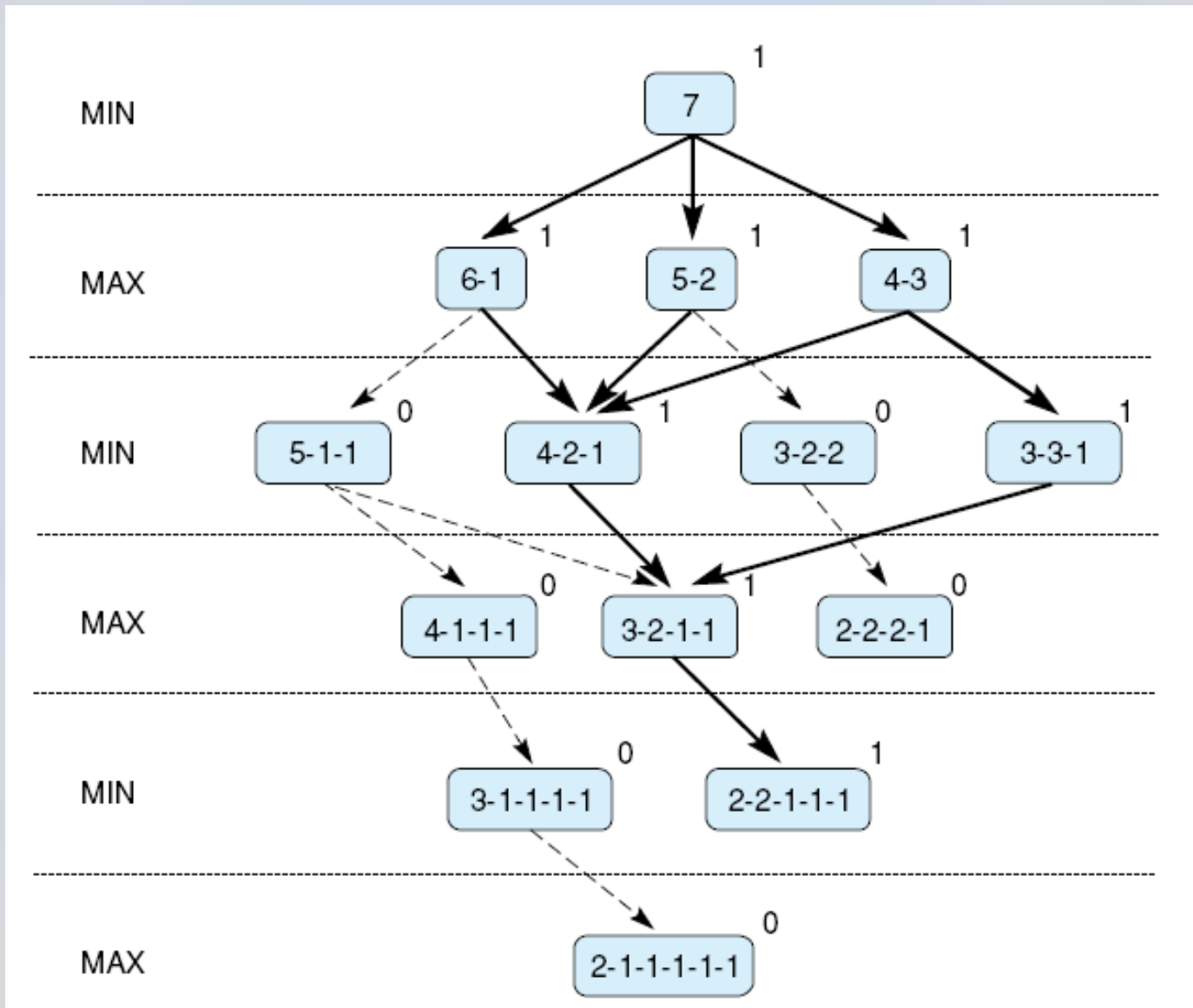
- osnova: igra dveh nasprotnikov
- primer nim: nasprotnika izmenoma delita ploščice na dva neenaka kupa; nasprotnik, ki ne more deliti izgubi
- primer: nim 7 (majhen prostor stanj, lahko pregledamo vse)



Princip MINIMAKS

- ✿ predpostavka: nasprotnika imata na voljo enake informacije, oba poskušata zmagati
- ✿ igralec MAX (poskuša maksimizirati svoj rezultat) proti igralcu MIN (poskuša minimizirati MAX-ov rezultat)
- ✿ vozlišča označimo po nivojih glede na to, kdo je na potezi
- ✿ liste označimo: 1 zmaga MAX, 0 zmaga MIN
- ✿ minimax vrednosti propagira navzgor
 - ✗ če je predhodnik MAX, dobi maksimum naslednikov
 - ✗ če je predhodnik MIN, dobi minimum naslednikov
 - ✗ vrednosti določajo, kaj največ lahko igralec pričakuje, in poteze izbiramo glede na njih

Minimaks na nim 7



Minimax fiksne globine

- ✱ v zanimivih igrah prostora stanj ne moremo pregledati do listov
- ✱ pregledamo do neke (fiksne) globine glede na čas, računske in pomnilniške zmogljivosti
- ✱ pogled naprej globine n (n -ply look-ahead)
- ✱ na globini n vozliščem določimo hevristično oceno kvalitete in jo propagiramo navzgor
- ✱ minimax tako izračuna oceno najboljšega stanja do globine n
- ✱ hevristike za šah, npr. število in moč figur, postavitev figur, ...

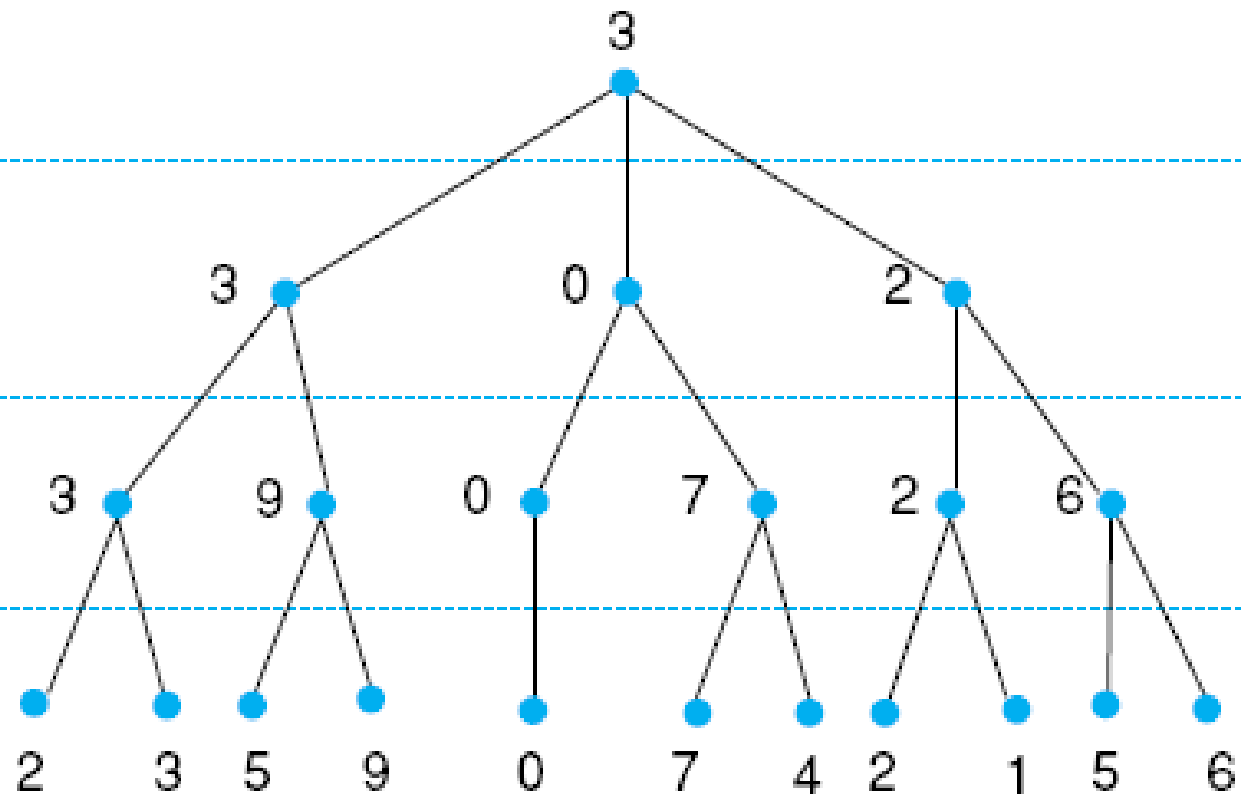
Primer: minimax globine 4

MAX

MIN

MAX

MIN



Minimax koda

```
double minimax (currentNode) {  
    if ( isLeaf (currentNode) || depth(currentNode) == MaxDepth )  
        return heuristicEvaluation (currentNode);  
  
    if ( isMinNode (currentNode) )  
        return min ( minimax (children (currentNode)) );  
  
    if ( isMaxNode (currentNode) )  
        return max (minimax (children (currentNode)) );  
}
```

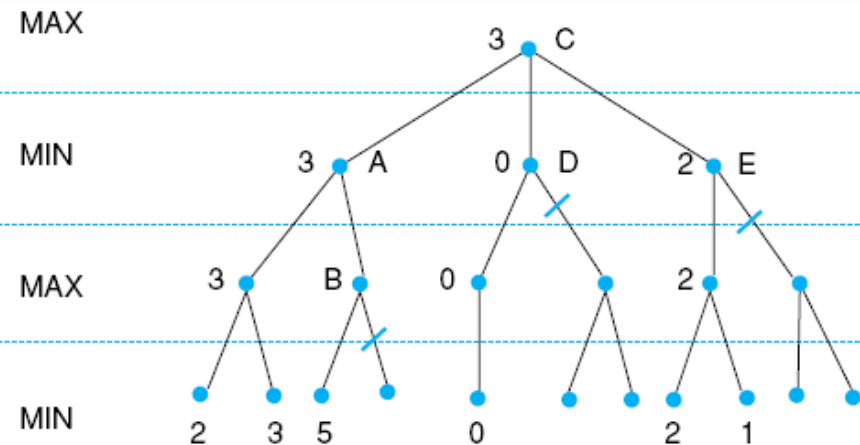
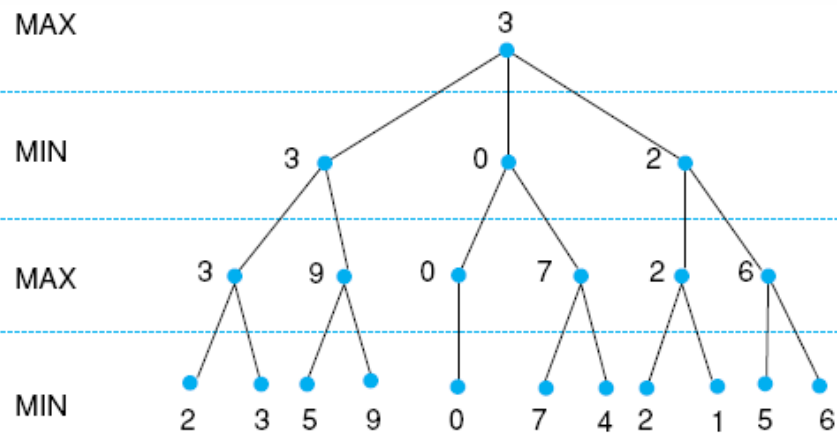
Analiza minimaksa

- ✱ kratkovidno, ne vidi dlje kot n
- ✱ (na videz) dobro stanje na neki globini lahko zapelje igralca
- ✱ popravek: selektivno preiščemo še nekaj nivojev naprej od dobrega stanja
- ✱ anomalije minimaksa: kršenje predpostavk, težavnost nekaterih pozicij

Alfa-beta rezanje

- ✿ minimaks preišče vsa vozlišča do globine n in ocene propagira navzgor
- ✿ številna poddrevesa so lahko neperspektivna in jih ne bi bilo treba preiskati
- ✿ ideja:
 - ✗ preiskujemo v globino,
 - ✗ za MAX vozlišča predstavlja α najboljšo doslej najdeno vrednost
 - ✗ za MIN vozlišča predstavlja β najslabšo vrednost doslej
 - ✗ zavržemo MAX vrednosti manjše od α in MIN vrednosti večje od β

Prikaz α - β rezanja



A has $\beta = 3$ (A will be no larger than 3)
 B is β pruned, since $5 > 3$
 C has $\alpha = 3$ (C will be no smaller than 3)
 D is α pruned, since $0 < 3$
 E is α pruned, since $2 < 3$
 C is 3

Pravila α - β rezanja

- ✱ odrežemo naslednike MIN vozlišča z β vrednostjo manjšo ali enako α vrednosti njegovega MAX predhodnika
- ✱ odrežemo naslednike MAX vozlišča z α vrednostjo večjo ali enako β vrednosti njegovega MIN predhodnika
- ✱ primerjamo torej nivo $m-1$ in $m+1$, da režemo na nivoju m

α - β rezanje: pedagoška implementacija

```
double alphaBeta (currentNode) {  
    if ( isLeaf (current_node) )  
        return heuristicEvaluation (currentNode);  
  
    if ( isMaxNode (currentNode) &&  
        alpha (currentNode) >= beta (minAncestor (currentNode) )  
        stopSearchBelow (currentNode);  
  
    if ( isMinNode (currentNode) &&  
        beta (currentNode) <= alpha (maxAncestor(currentNode) )  
        stopSearchBelow (currentNode);  
  
}
```

α - β rezanje: implementacija

- v izogib doseganju vrednosti α in β nazaj pri predhodnikih, te vrednosti pošljemo kot parametre
- za MAX vozlišče shranimo minimum β vrednosti svojih MIN naslednikov kot beta
- za MIN vozlišče shranimo maksimum α vrednost svojih MAX naslednikov kot alpha
- vsako notranje vozlišče bo tako hranilo vrednosti alpha in beta
- koren drevesa inicializiramo kot $\alpha = -\infty$, $\beta = \infty$

α - β rezanje: koda

```
double alpha_beta (current_node, alpha, beta) {  
    if ( is_leaf (current_node) )  
        return heuristic_evaluation (current_node);  
    if ( is_max_node (current_node) ) {  
        alpha = max (alpha, alpha_beta (children, alpha, beta));  
        if alpha >= beta  
            cut_off_search_below (current_node);  
    }  
    if is_min_node (current_node){  
        beta = min (beta, alpha_beta (children, alpha, beta));  
        if beta <= alpha  
            cut_off_search_below (current_node);  
    }  
}
```

✱ klic: alpha_beta (start_node, -infinity, infinity)

Dama

- ✿ $(b \approx 8, n \approx 10^{20})$
- ✿ (Samuel, 1959)
 - ✗ minimax, α - β , ocena pozicij
 - ✗ na nivoju srednje dobrih igralcev
- ✿ Chinook (Schaeffer, 1990, 1994-)
 - ✗ minimax, α - β , zelo dobra ocena pozicij, baza končnic z do 8 figur, baza otvoritev, do globine 20, rezanje vnaprej glede na izgubo
 - ✗ premaga svetovnega prvaka
- ✿ Blondie24 (Fogel, 2000)
 - ✗ uči se strategije z nevronskimi mrežami in evolucijskimi pristopi
 - ✗ na ravni dobrih igralcev

Šah

✻ $(b \approx 38, n \approx 10^{120})$

- ✻ minimax, α - β , ocena pozicij, baza otvoritev
- ✻ najboljši programi preiskujejo do globine približno 12
- ✻ linearna povezava med globino preiskovanja in kvaliteto igre
- ✻ 1997, DeepBlue premagal svetovnega prvaka
- ✻ DeepFritz, Rybka

Nekaj drugih iger

- ✱ Go: plošča 19×19 , $b \approx 360$, na nivoju povprečnih igralcev
- ✱ Go-Moku, 15×15 , varianta 5 v vrsto, dokončno rešena
- ✱ Othello (reversi), do globine 50, premaga svetovnega prvaka
- ✱ arimaa, šahovska plošča, figure ($b \approx 17281$)
- ✱ verjetnostne igre: backgammon, bridge, tarok, poker
- ✱ expectiminimax: verjetnostna inačica minimaksa
- ✱ Monte-Carlo drevesno preiskovanje: selektivno naključno preiskovanje nekaterih pozicij, statistična ocena uspešnosti, dilema med raziskovanjem novih potez in izkoriščanjem že znanih dobrih potez

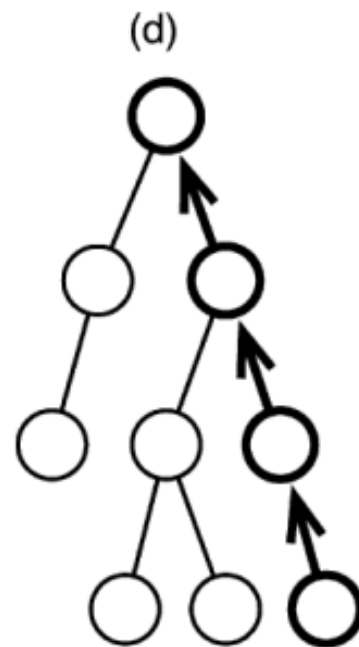
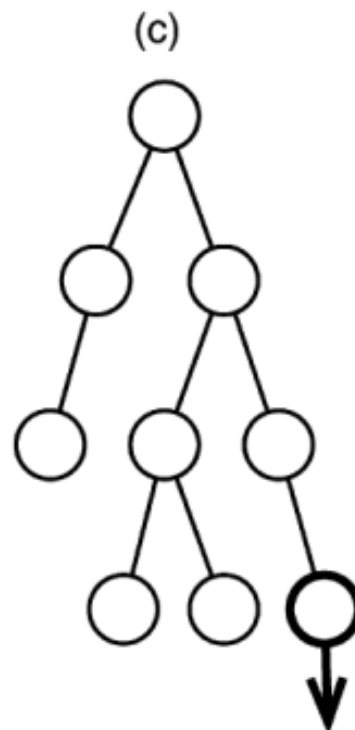
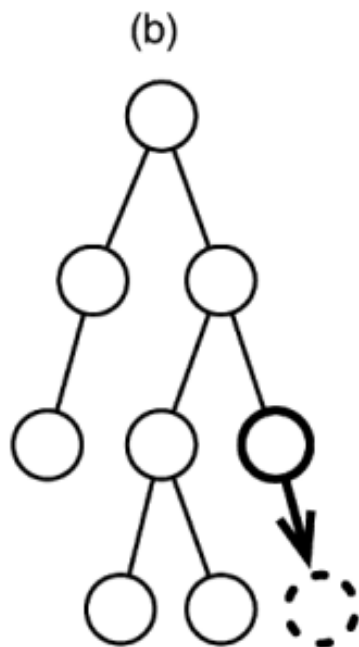
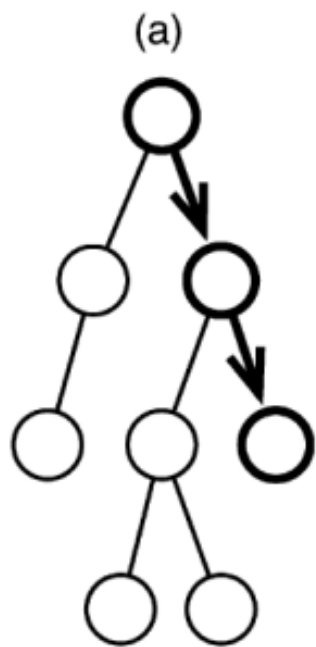
Expectiminimax

- ✱ kot minimax, a v uporabi tam, kjer so izidi negotovi, npr. v igrah z verjetnostnimi elementi (karte, kocke, igre na srečo....)
- ✱ namesto determinističnega minimaxa izračunaj pričakovani minimax (matematično upanje)
- ✱ hevristične ocene ocenijo verjetnost dogodkov
- ✱ primer: oceni verjetnost, da ima igralec v roki določeno karto

Drevesno preiskovanje Monte Carlo

Monte Carlo Tree Search (MCTS)

- za velike prostore stanj so izčrpane metode neuporabne
- ocene kakovosti stanj je včasih težko določiti, potrebno je razviti teorijo za vsako igro posebej
- alternativa.: oceni stanje na podlagi vzorčenja
- predstavi preiskovalni prostor z drevesom
- ideja:
 1. simuliraj naključne ocene, dokler se igra ne konča (torej do lista drevesa)
 2. propagiraj odločitev proti korenu
 3. ponavljaj postopek
 4. izberi vozlišče (potezo)) glede na delež uspešnih iger
- ocenjevalna heuristika ni potrebna
- MCTS dokazano konvergira v minimax, ko gre število simulacij v neskončnost



Sestavni deli MCTS

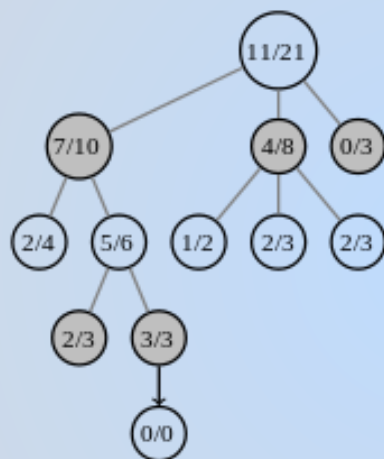
- ✿ mnogo naključnih simulacij
- ✿ štirje iterativni koraki
 - ✿ **izbira** – izberi (vrhnje) vozlišče
 - ✿ **širitev** – dodaj nova vozlišča v drevo
 - ✿ **simulacija** – simuliraj igro iz izbranega vozlišča
 - ✿ **ažuriranje** – rezultat iz lista se propagira proti korenu, obiskana vozlišča dobijo nove vrednosti

Ilustracija MCTS

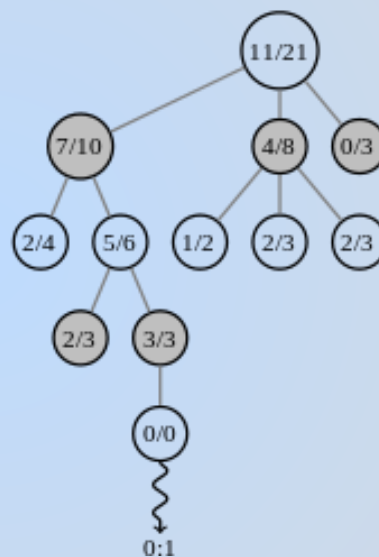
Selection



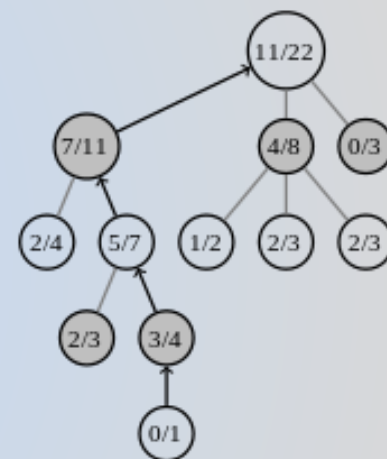
Expansion



Simulation



Backpropagation



“Hevristična ocena” vozlišč

- ✱ v resnici izbira vozlišča
- ✱ popolnoma naključno – mogoče, a je variance velika
- ✱ najdi ravnotežje med “išči ali izkoristi” (explore or exploit)
- ✱ način UCT (Upper Confidence bounds applied to Trees)

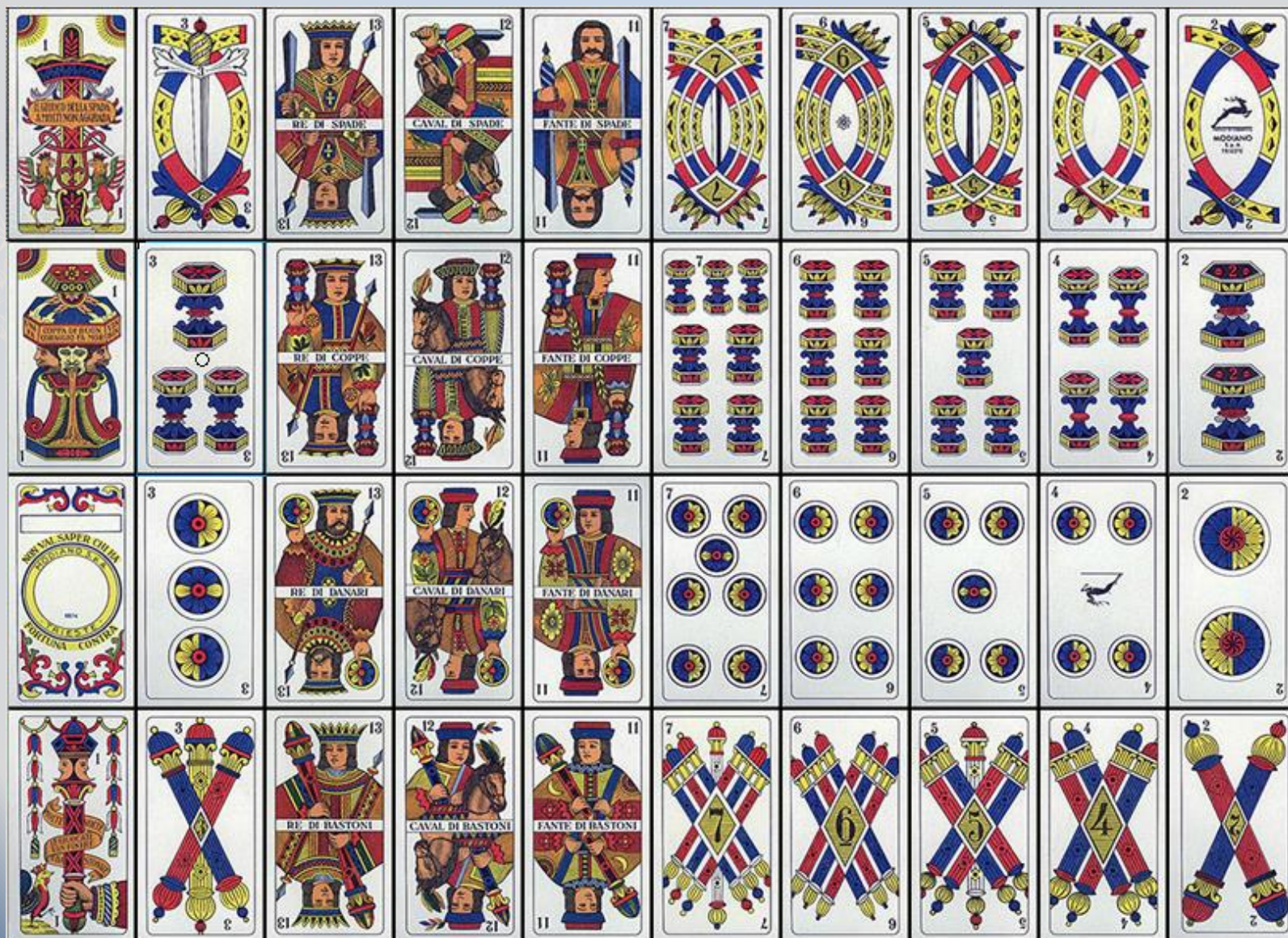
UCT

$$k = \operatorname{argmax}_{i \in I} \left(\frac{w_i}{n_i} + C \sqrt{\frac{\ln n}{n_i}} \right)$$

- izračunaj UCT za vsa vrhnja (kandidatna) vozlišča i (poteze), kjer je
 w_i – število zmag iz vozlišča i
 n_i – število obiskov vozlišča i
 n – število obiskov vseh vrhnjih vozlišč $n = n_1 + n_2 + \dots$
 C – koeficient za uravnavanje izbire med išči ali izkoristi

izberi vozlišče (potezo) z maksimalnim UCT

Praktičen primer: igra tršet



Pravila tršeta

- ☀ 2, 3 ali 4 igralci
- ☀ 4 barve po 10 kart
- ☀ moč kart (najmočnejša jemlje)
 - ✂ 3, 2, as, kralj, kaval, fant
 - ✂ 7, 6, 5, 4 (*brez točk*)
- ☀ točkovanje
 - ✂ as: 1 točka
 - ✂ 3, 2, kralj, kaval, fant: $\frac{1}{3}$ točke
 - ✂ 7, 6, 5, 4: 0 točk
 - ✂ zadnji vzetek: 1 točka
- ☀ pravilo odgovarjanja na barvo

Tršet za dva igralca

- ✿ vsak igralec dobi 10 kart
- ✿ 20 kart ostane na kupu s hrbtom obrnjenim navzgor (element negotovosti)
- ✿ po vsakem vzetku igralca vzameta po eno karto iz kupa in jo pokažeta nasprotniku
- ✿ postopno se negotovost zmanjšuje
- ✿ po 10 rundi imamo na voljo vso informacijo

Tipične strategije tršeta

- ✿ skupaj je na voljo 11 in $\frac{2}{3}$ točke, za zmago jih je potrebno dobiti 6
- ✿ preproste heuristike:
 - ✗ igralci skušajo osvojiti svoje in nasprotnikove ase, 2, 3 in glave (kralj, kaval, fant)
 - ✗ skušajo dobiti zadnjo rundo
 - ✗ prednost so sekvence

Avtomatski igralec za tršet v dva

- ✱ cilj: razviti avtomatskega igralca tršeta na osnovi expectiminimaksa in MCTS
- ✱ časovna omejitev: 1 sekunda računskega časa na strežniku za vsako potezo
- ✱ magistrsko delo Žana Kafola

Expectiminimax

- ✱ v vsaki rundi
- ✱ generiraj vse možne konfiguracije nasprotnikovih kart in kart v talonu
- ✱ preveč kombinacij za izčrpno preiskovanje
- ✱ ustavi se na nekem nivoju in hevristično oceni situaciji
- ✱ težko je najti hevristiko za oceno kakovosti igralne pozicije

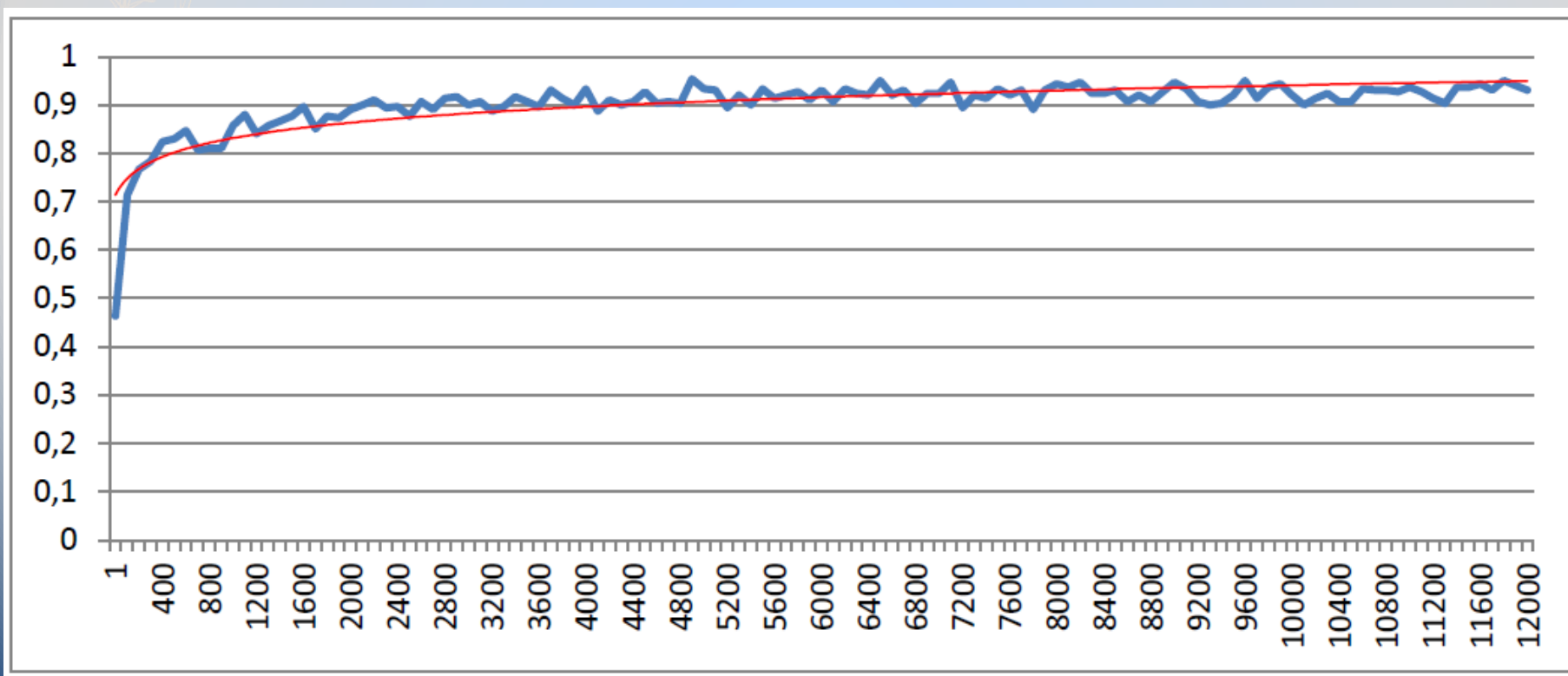
Minimax

- ✱ od 10. runde naprej imamo na voljo vse informacije
- ✱ lahko iščemo izčrpno z minimaksom
- ✱ uporabimo α - β rezanje

MCTS

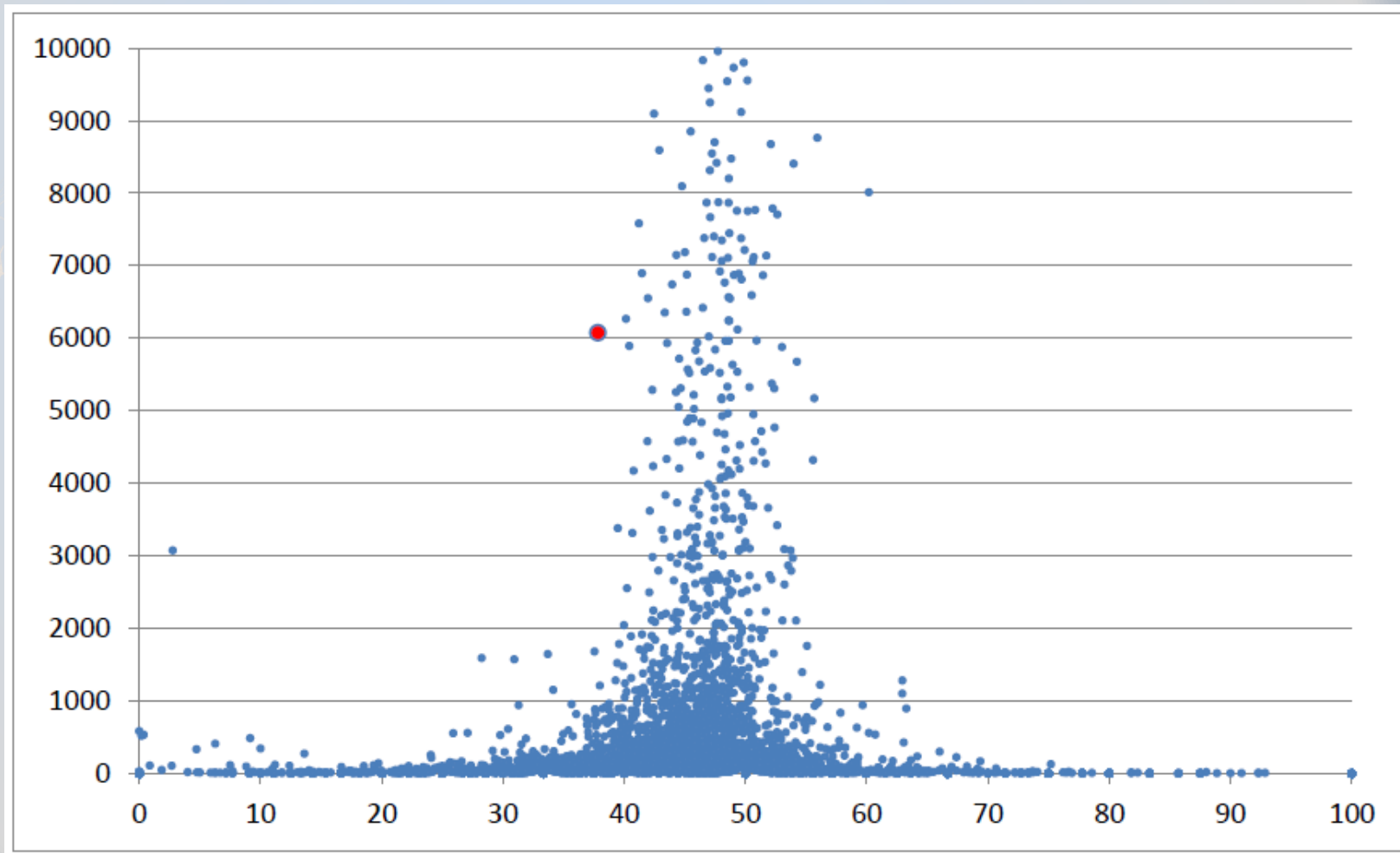
- ✱ do 10. runde, ko smo še v negotovosti in bi bilo vseh kombinacij preveč
- ✱ uporabimo MCTS z UCT izbiro
- ✱ upoštevamo časovno omejitev

MCTS proti naključnemu igralcu



MCTS proti človeškim igralcem

- ✱ 37.7% uspeh, povprečen človeški igralec 38.9%



Izboljšave MCTS

- ✱ ideja: uporabi shranjene poteze človeških igralcev
- ✱ določi začetno kakovost posameznih potez
- ✱ izboljšaj z MCTS
- ✱ majhna verjetnost, da bi našli povsem enako igro
- ✱ problem: kako določiti podobne igre?

Pristop k izrabi človeškega znanja

- ✱ 16 milijonov shranjenih potez
- ✱ uporabi le dobre (nadpovprečne) igralce
- ✱ določi attribute igre, da najdeš podobne igre
- ✱ npr. število asov, dvojok, trojk, sekvence, ...
- ✱ najdi poteze človeških igralcev narejene v podobnih okoliščinah
- ✱ za iskanje uporabi npr. kd-drevesa

Uporaba človeškega znanja

☀ 1. način

- ✂ uporabi kNN za izbiro najboljše poteze glede na dano strukturo podobnosti
- ✂ izbrano potezo prilagodi na konkretno situaciji

☀ 2. način

- ✂ preštej zmage glede na dano potezo in uporabi to v UCT

- ☀ le majhen napredek v primerjavi z osnovnim MCTS: na voljo ni dovolj časa, da bi naredil dovolj MCTS iteracij