COMP 472: Artificial Intelligence
Machine Learning part 2
Neural Networks video #8

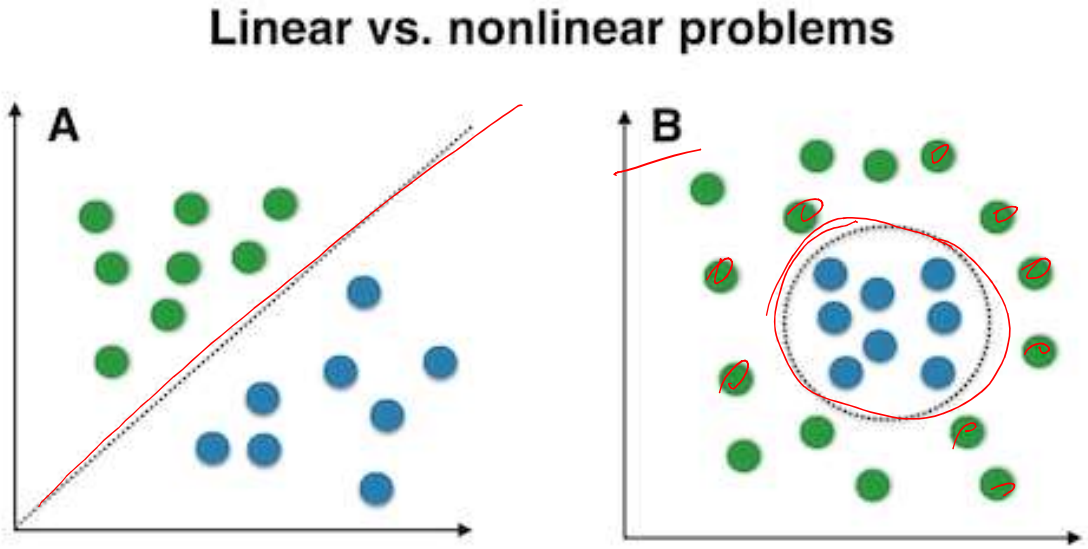
- Russell & Norvig: Sections 19.6, 21.1

Today

1. Introduction to ML
2. Naïve Bayes Classification
   a. Application to Spam Filtering
3. Decision Trees
4. (Evaluation
5. Unsupervised Learning)
6. Neural Networks
   a. Perceptrons
   b. Multi Layered Neural Networks

Limits of the Perceptron

- can only model linear decision boundaries
- but real-world problems cannot always be represented by linearly-separable functions…

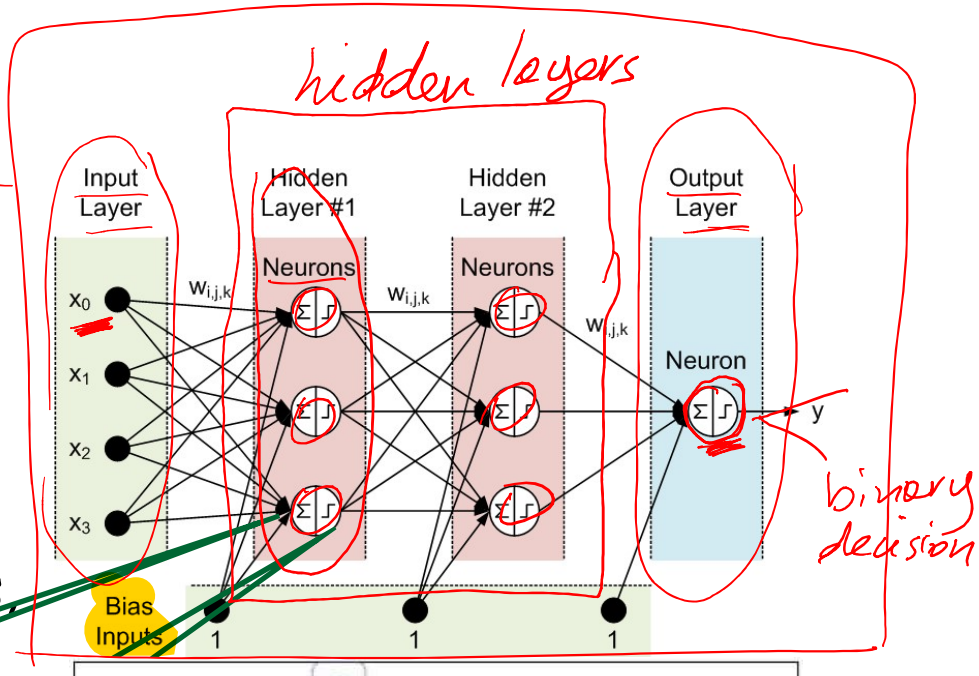


http://sebastianraschka.com/Articles/2014_kernel_pca.html

Multilayer Neural Networks

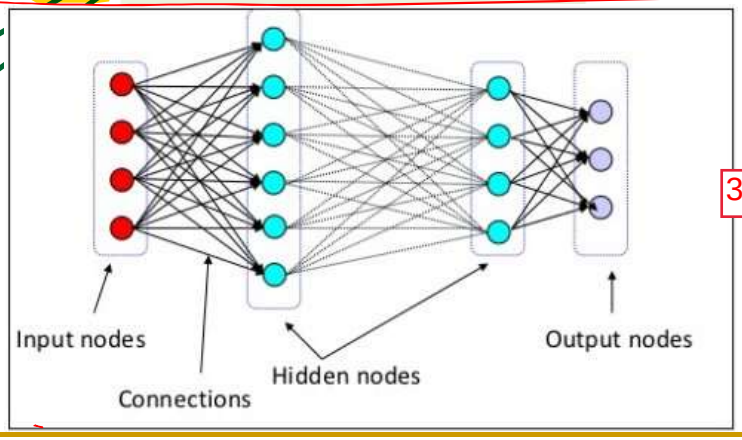
- Solution:

1. use a non-linear activation function
2. to learn more complex functions (more complex decision boundaries), have hidden nodes
3. and for non-binary decisions, have multiple output nodes



usual transfer function

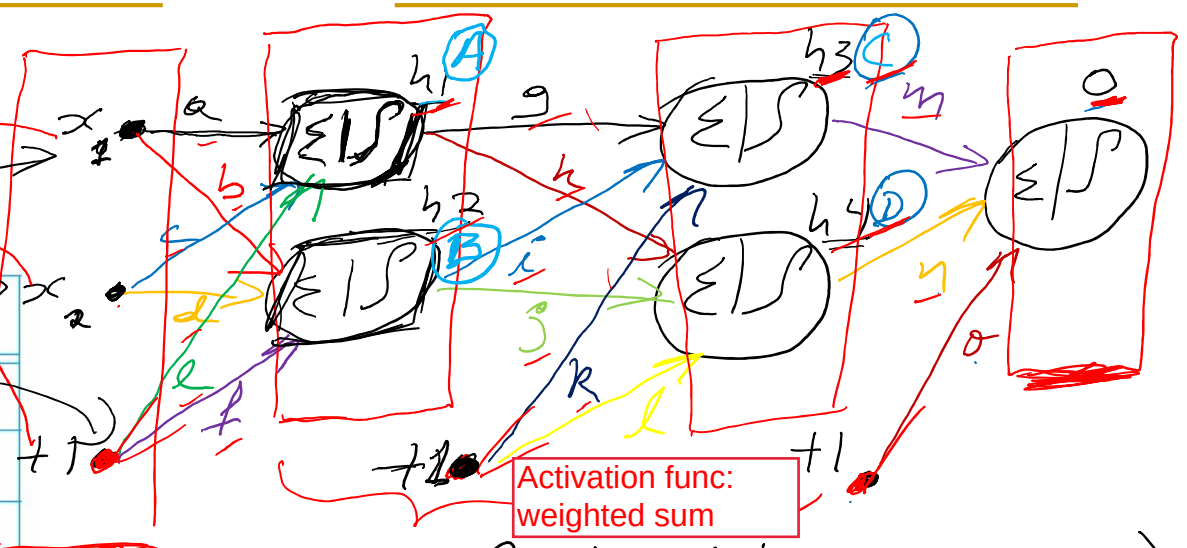
Non-linear, differentiable activation function



3 classes

http://www.kdnuggets.com/2016/10/deep-learning-key-terms-explained.html

Example



x_1	x_2	bias	Output (Target)
1.0	1.0	1	1
9.4	6.4	1	−1
2.5	2.1	1	1
8.0	7.7	1	−1
0.5	2.2	1	1
7.9	8.4	1	−1
7.0	7.0	1	−1
2.8	0.8	1	1
1.2	3.0	1	1
7.8	6.1	1	−1

output of $h1 = \int(\varepsilon) = \int(x_1 \cdot a + x_2 \cdot c + 1 \cdot e) = A$

output of $h2 = \int(x_1 \cdot b + x_2 \cdot d + 1 \cdot f) = B$

output of $h3 = \int(A \cdot g + B \cdot i + 1 \cdot k) = C$

output of $h4 = \int(A \cdot h + B \cdot j + 1 \cdot l) = D$

output of $O = \int(C \cdot m + D \cdot n + 1 \cdot o) = E$

feed forward

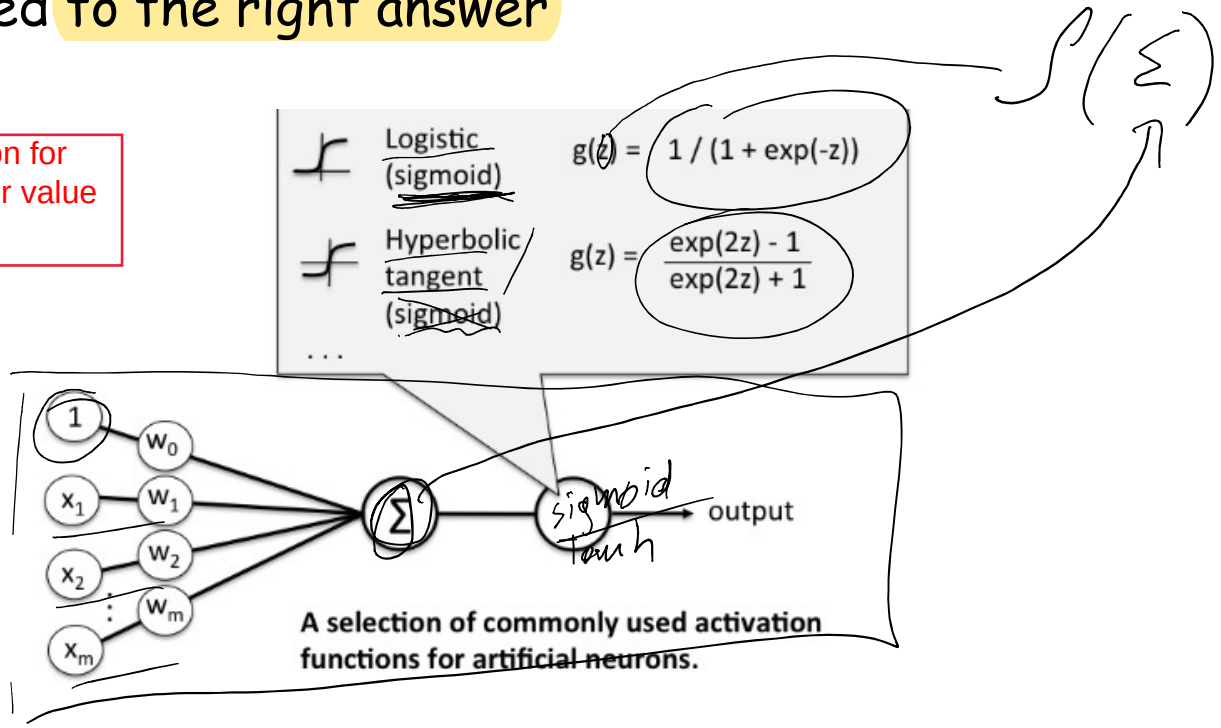
backprop

output of the network

Activation Functions

- Backpropagation requires a differentiable activation function
- that returns continuous values within a range
  - eg. a value between 0 and 1 (instead of 0 or 1, like the perceptron)
- indicates how close/how far the output of the network is compared to the right answer

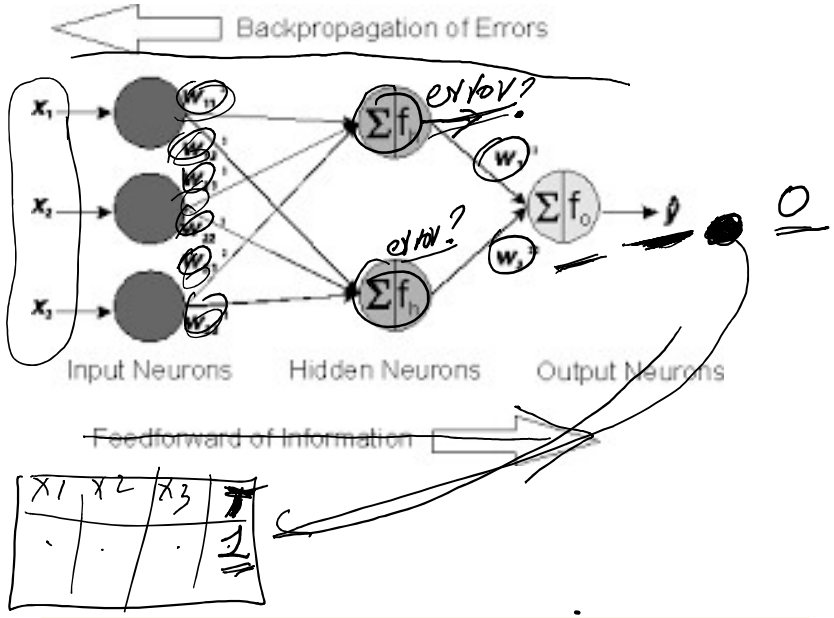
Sigmoid function for calculating error value



https://medium.com/towards-data-science/activation-functions-and-its-types-which-is-better-a9a5310cc8f

Learning in a Neural Network

- Learning is the same as in a perceptron:



1. Feed-forward:
   - Input from the features is fed forward in the network from input layer towards the output layer

2. Backpropagation:
   - Error rate flows backwards from the output layer to the input layer (to adjust the weights in order to minimize the error)

3. Iterate until error rate is minimized
   - repeat the forward pass and back pass for the next data points until all data points are examined (1 epoch)
   - repeat this entire exercise (several epochs) until the overall error is minimised

Typical Cost Functions

- Error of the network is computed via a cost function

1. Quadratic Cost:
   - aka mean squared error (MSR)
   - minimize the difference between output values and target values

$$C = \frac{1}{n}\sum_{i=1}^{n}(T_i - O_i)^2$$

where n = nb of instances

$i=1 \quad (T_1 - O_1)^2$
$i=2 \quad (T_2 - O_2)^2$
$i=3 \quad (T_3 - O_3)^2$
$\frac{1}{3}\Sigma \sim\sim$

   - used mostly for regression tasks

2. Cross-entropy:
   - used mostly for classification tasks
   - where we don't care about the exact value of the network output, we only care about the final class

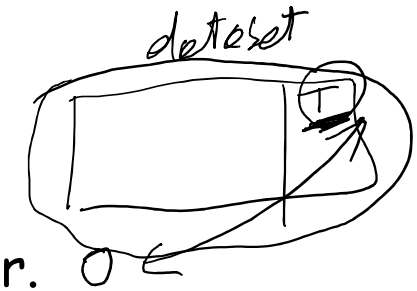
$$C = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\left(T_{ik}\,\log(O_{ik})\right)$$

where:
n = nb of instances
K = nb of classes
log = base 2, base 10, ln, ...

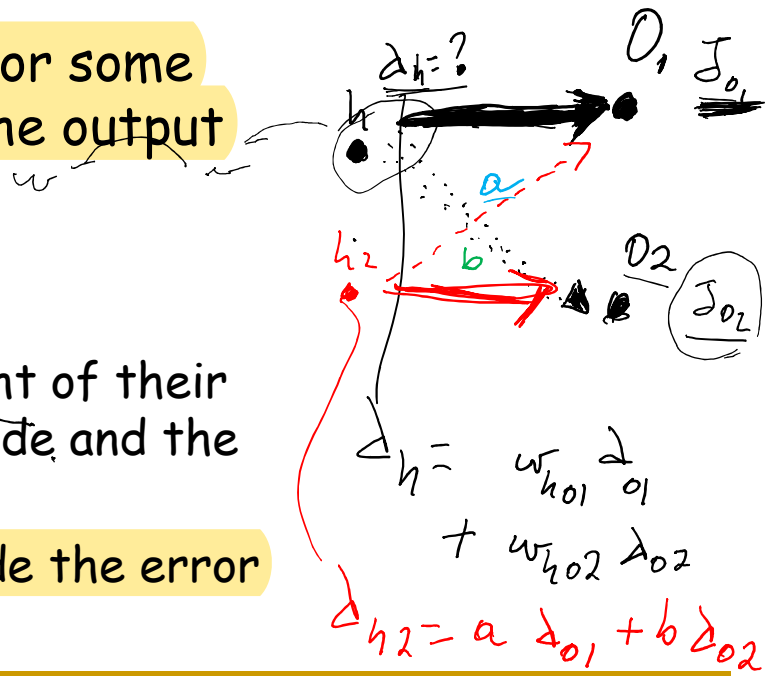
Example of Cross Entropy

- Assume 3 classes: red, blue and green
  - i.e. $K = 3$
- for a specific instance i, the target is green
  - ie. (red, blue, green) = (0, 0, 1) // real distribution
  - $T^1 = 0 \; T^2 = 0 \; T^3 = 1$
- but the model predicts the probabilities as:
  - (red, blue, green) = (0.1, 0.4, 0.5) // predicted distribution
  - $O^1 = 0.1 \; O^2 = 0.4 \; O^3 = 0.5$
- the cross entropy of two distributions (real and predicted)
  - $-\sum_{k=1}^{K} \left(T^k \ln(O^k)\right) = -((0)ln(0.1) + (0)ln(0.4) + (1)ln(0.5))$
- but we have several instances in the test set, so we will take the average of the cross-entropy across all instances i in the test set
  - $C = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}(T^k{}_i \ln(O^k{}_i))$
- see an example calculation computation in a few slides

Backpropagation

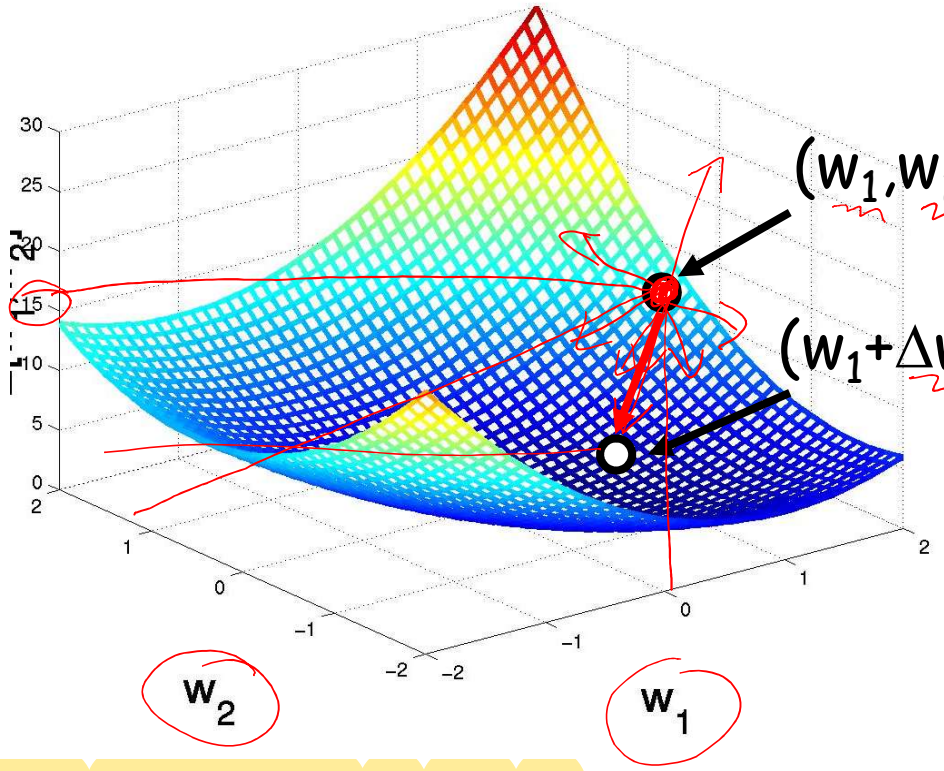


- In a multilayer network…
  - Computing the error in the output layer is clear.
  - Computing the error in the hidden layer is not clear, because we don't know what output it should be
- Intuitively:
  - A hidden node h is "responsible" for some fraction of the error in each of the output node to which it connects.
  - So the error values (δ):
    - are divided according to the weight of their connection between the hidden node and the output node
    - and are propagated back to provide the error values (δ) for the hidden layer.



Gradient Descent - Adjusting the Weights

$C(w_1, w_2)$



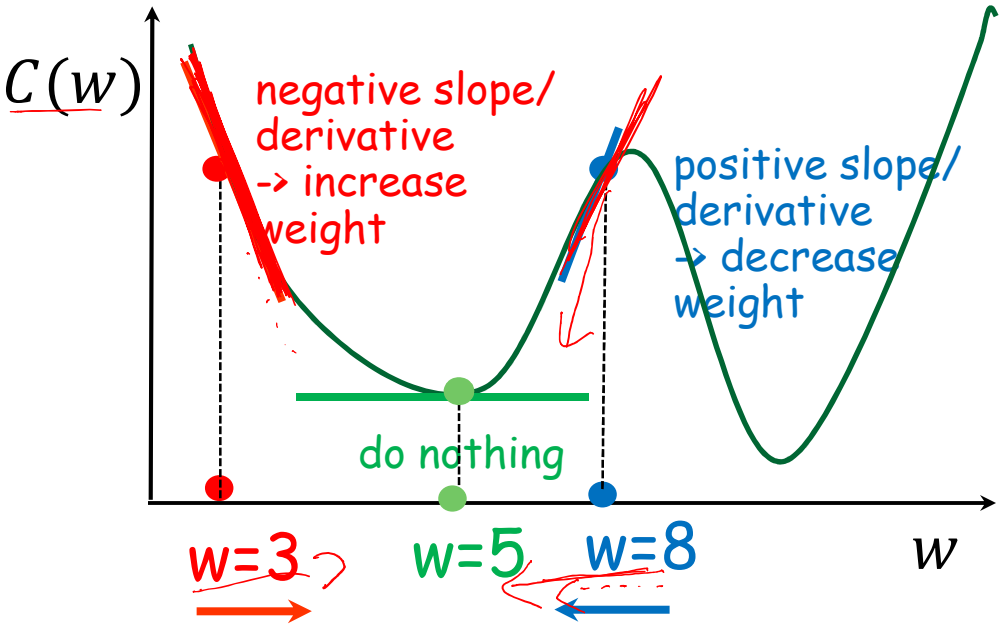
- Goal: minimize $C(w_1, w_2)$ by changing w_1 and w_2
- What is the best combination of change in w_1 and w_2 to minimize C faster?

Gradients

Gradient is just derivative in 1D

Eg: $C(w) = (w-5)^2$ derivative is: $\frac{\partial C}{\partial w} = 2(w-5)$

only 1 weight



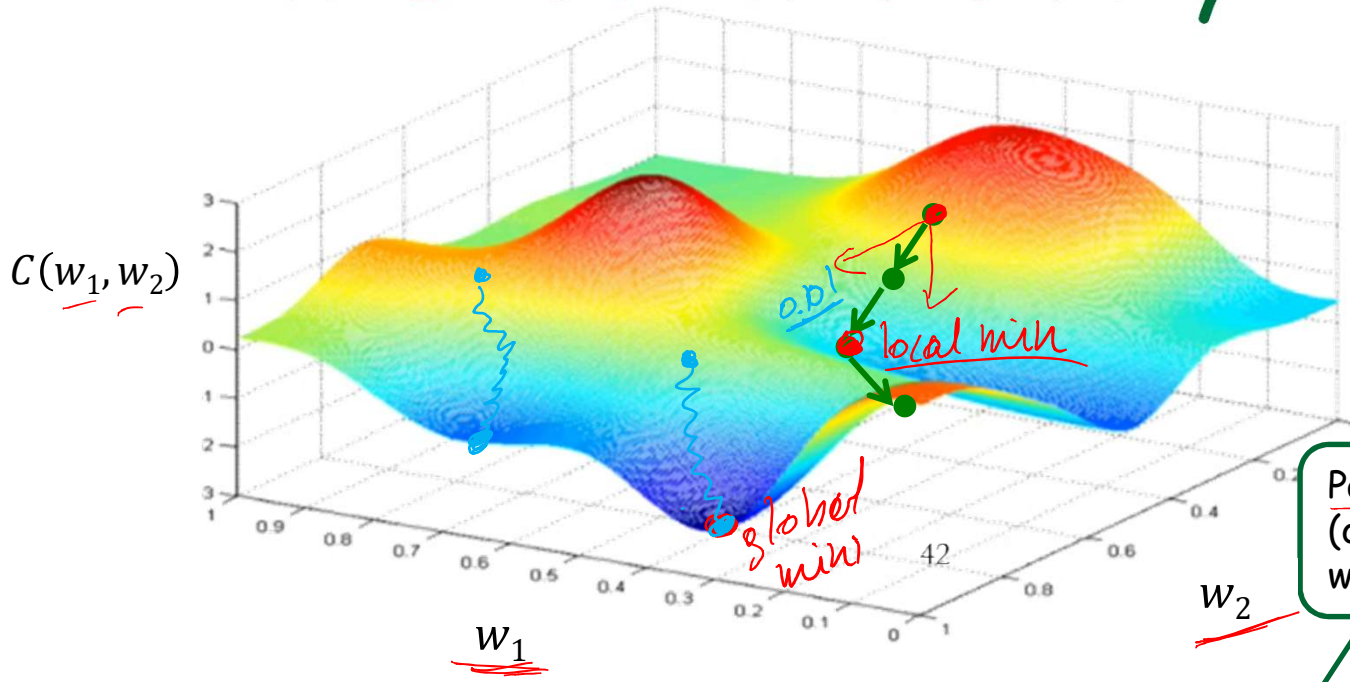
If w=3 $\frac{\partial C}{\partial w}(3) = 2(3-5) = -4$

derivative says increase w (go in opposite direction of derivative)

If w=8 $\frac{\partial C}{\partial w}(8) = 2(8-5) = 6$

derivative says decrease w (go in opposite direction of derivative)

Gradient Descent Visually



Partial derivative (or gradient) of C with respect to w1

- need to know how much a change in w1 will affect C(w1,w2) i.e $\frac{\partial C}{\partial w_1}$
- need to know how much a change in w2 will affect C(w1,w2) i.e $\frac{\partial C}{\partial w_2}$
- Gradient ∂C points in the opposite direction of steepest decrease of C(w1,w2)
- i.e. hill-climbing approach…

Source: Andrew Ng

Training the Network

After some calculus (see: https://en.wikipedia.org/wiki/Backpropagation we get…

- Step 0: Initialise the weights of the network randomly

// feedforward

- Step 1: Do a forward pass through the network

$$O_i = g\left(\sum_j w_{ji} x_j\right) = \text{sigmoid}\left(\sum_j w_{ji} x_j\right) = \frac{1}{1+e^{-\left(\sum_j w_{ji} x_j\right)}}$$

Note: To be consistent with Wikipedia, we'll use O-T instead of T-O but we will subtract the error in the weight update

// propagate the errors backwards

- Step 2: For each output unit k, calculate its error term δ_k

$$\delta_k \leftarrow g'(x_k) \times Err_k = O_k(1 - O_k) \times (O_k - T_k)$$

Derivative of sigmoid

note, if we use $g = \text{sigmoid}$:

$$g'(x) = g(x)(1 - g(x))$$

- Step 3: For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow g'(x_h) \times Err_h = O_h(1 - O_h) \times \sum_{k \in \text{outputs}} w_{hk} \delta_k$$

Sum of the weighted error term of the output nodes that h is connected to (ie. h contributed to the errors δ_k)

- Step 4: Update each network weight w_{ij}:

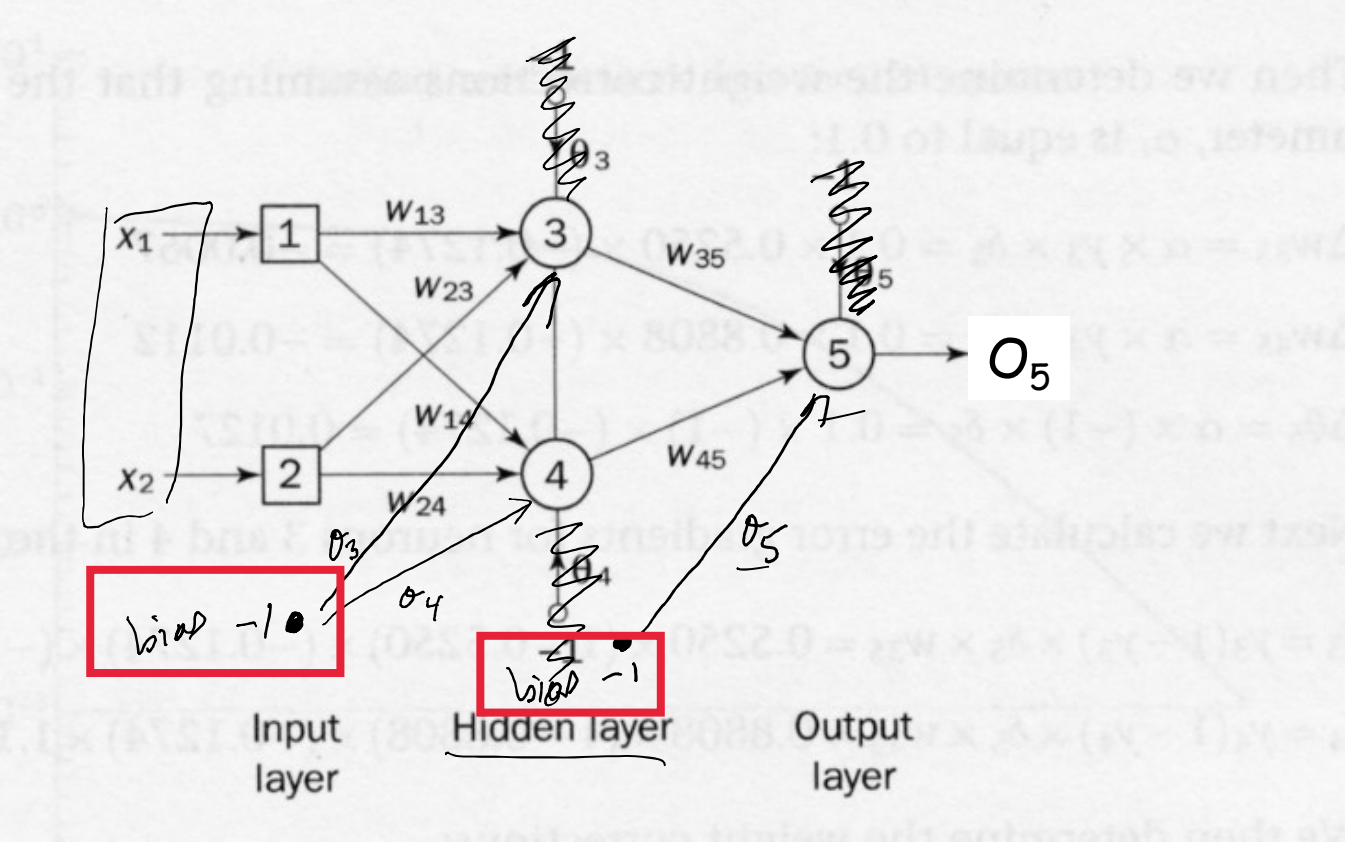
$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{where } \Delta w_{ij} = -\eta \delta_j O_i$$

$\equiv x_i$ in slide #22

- Repeat steps 1 to 4 until the cost (C) is minimised

Example: XOR

x_1 XOR x_2

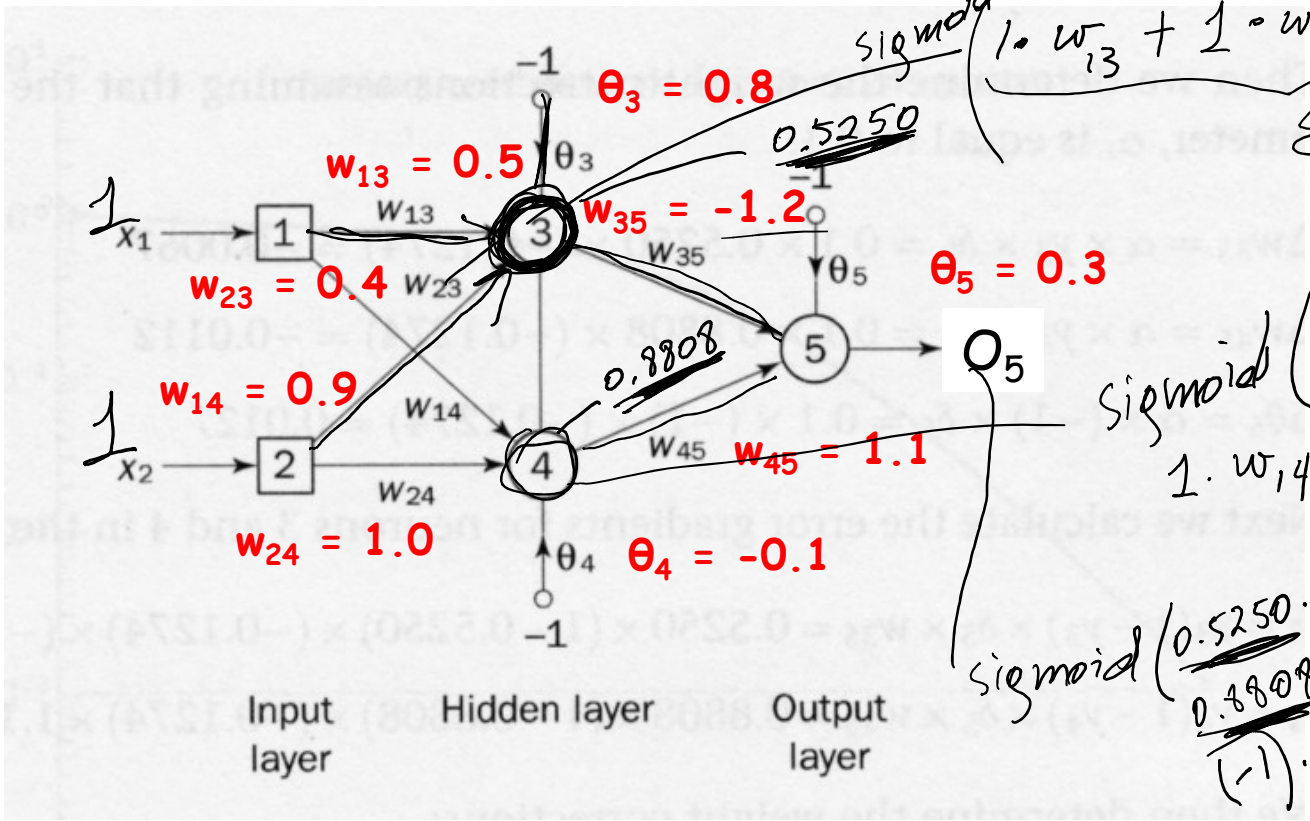


- 2 input nodes + 2 hidden nodes + 1 output node + 3 biases

source: Negnevitsky, Artificial Intelligence, p. 181

Example: Step 0 (initialization)

- Step 0: Initialize the network at random



Step 1: Feed Forward

- Step 1: Feed the inputs and calculate the output

$$O_i = \text{sigmoid}\left(\sum_j w_{ji} x_j\right) = \frac{1}{1 + e^{-\left(\sum_j w_{ji} x_j\right)}}$$

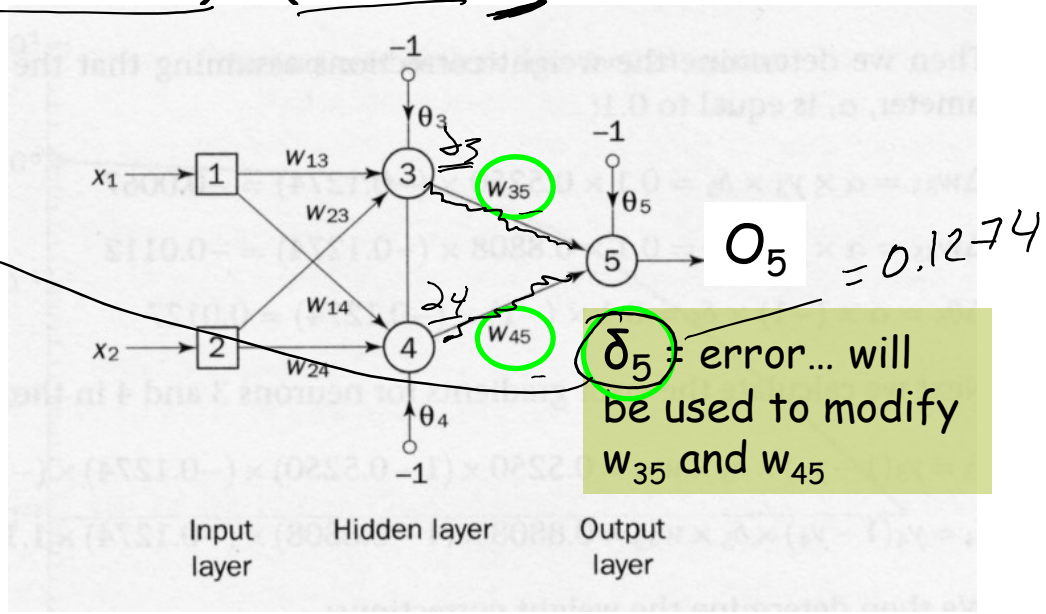
x_1	x_2	Target output T
1	1	0
0	0	0
1	0	1
0	1	1

  - With (x_1=1, x_2=1) as input:
    - Output of the hidden node 3:
      - O_3= sigmoid($x_1 w_{13} + x_2 w_{23} - \theta_3$) = 1 / (1 + $e^{-(1x.5 + 1x.4 - 1x.8)}$)= 0.5250
    - Output of the hidden node 4:
      - O_4= sigmoid($x_1 w_{14} + x_2 w_{24} - \theta_4$) = 1 / (1 + $e^{-(1x.9 + 1x1.0 + 1x0.1)}$)= 0.8808
    - Output of neuron 5:
      - O_5 = sigmoid($O_3 w_{35} + O_4 w_{45} - \theta_5$) = 1 / (1 + $e^{-(0.5250x-1.2 + 0.8808x1.1 - 1x0.3)}$)= 0.5097

Step 2: Calculate error term of output layer

$$\delta_k \leftarrow g'(x_k) \times Err_k = O_k(1 - O_k) \times (O_k - T_k)$$

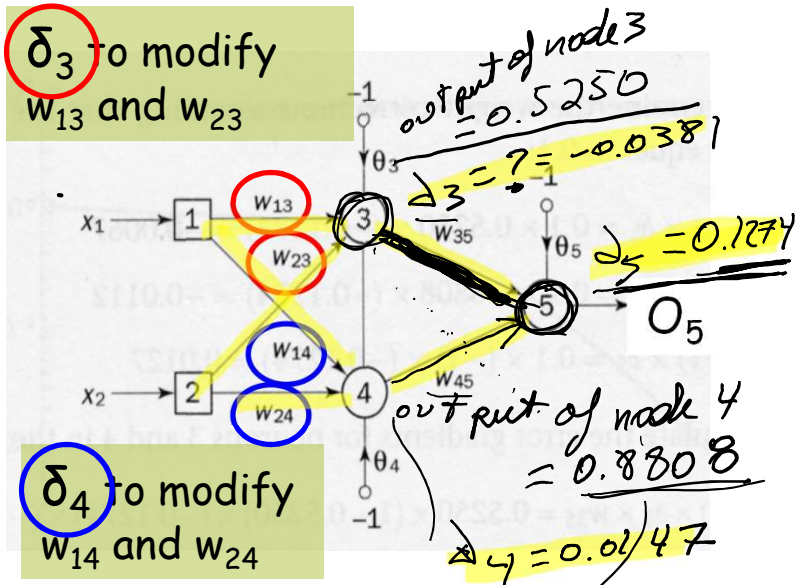
- Error term of neuron 5 in the output layer:
  - $\delta_5 = O_5 (1-O_5) (O_5 - T_5)$
    $= (0.5097) \times (1-0.5097) \times (0.5097-0)$
    $= 0.1274$



Step 3: Calculate error term of hidden layer

$$\delta_h \leftarrow g'(x_h) \times Err_h = O_k(1 - O_k) \times \sum_{k \in outputs} w_{kh}\delta_k$$

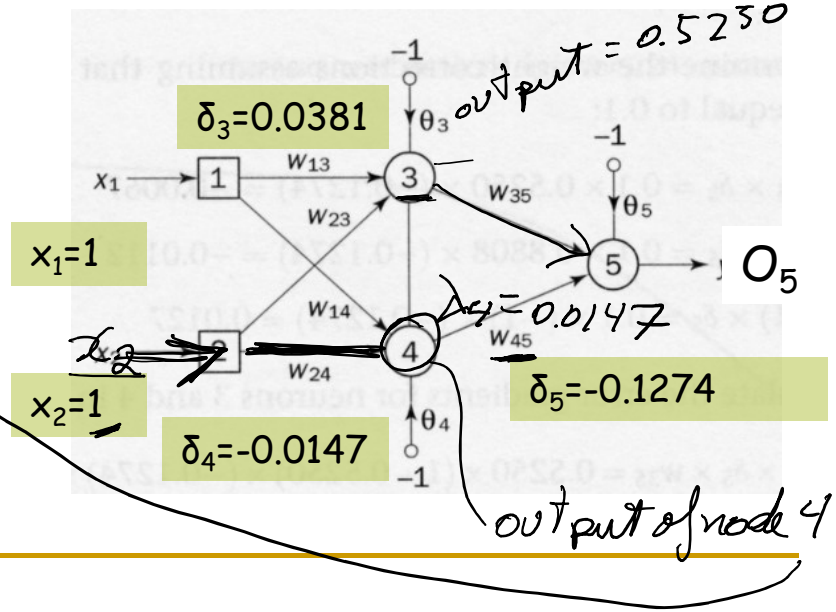
- Error term of neurons 3 & 4 in the hidden layer:
  - $\delta_3 = O_3 (1-O_3)\ \delta_5 w_{35}$
    $= (0.5250) \times (1-0.5250) \times (0.1274) \times (-1.2)$
    $= -0.0381$
  - $\delta_4 = O_4 (1-O_4)\ \delta_5 w_{45}$
    $= (0.8808) \times (1-0.8808) \times (0.1274) \times (1.1)$
    $= 0.0147$



Step 4: Update Weights

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{where } \Delta w_{ij} = -\eta \, \delta_j \, x_i$$

- Update all weights (assume a constant learning rate η = 0.1)
  - $\Delta w_{13} = -\alpha \, \delta_3 \, x_1 = -0.1 \times -0.0381 \times 1 = 0.0038$
  - $\Delta w_{14} = -\alpha \, \delta_4 \, x_1 = -0.1 \times 0.0147 \times 1 = -0.0015$
  - $\Delta w_{23} = -\alpha \, \delta_3 \, x_2 = -0.1 \times -0.0381 \times 1 = 0.0038$
  - $\Delta w_{24} = -\alpha \, \delta_4 \, x_2 = -0.1 \times 0.0147 \times 1 = -0.0015$
  - $\Delta w_{35} = -\alpha \, \delta_5 \, O_3 = -0.1 \times 0.1274 \times 0.5250 = -0.00669$ // O_3 is seen as x_5 (output of 3 is input to 5)
  - $\Delta w_{45} = -\alpha \, \delta_5 \, O_4 = -0.1 \times 0.1274 \times 0.8808 = -0.01122$ // O_4 is seen as x_5 (output of 4 is input to 5)
  - $\Delta \theta_3 = -\alpha \, \delta_3 \, (-1) = -0.1 \times -0.0381 \times -1 = -0.0038$
  - $\Delta \theta_4 = -\alpha \, \delta_4 \, (-1) = -0.1 \times 0.0147 \times -1 = -0.0015$
  - $\Delta \theta_5 = -\alpha \, \delta_5 \, (-1) = -0.1 \times 0.1274 \times -1 = -0.0127$

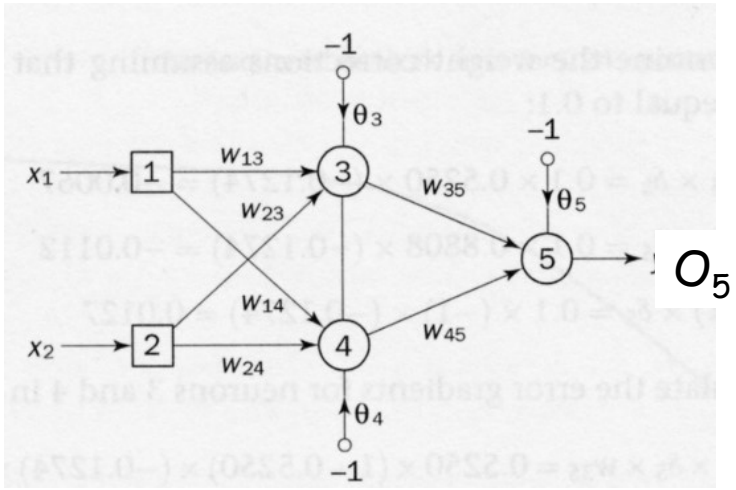


Step 4: Update Weights (con't)

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{where } \Delta w_{ij} = -\eta \, \delta_j \, x_i$$

- Update all weights (assume a constant learning rate η = 0.1)

  - $w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$
  - $w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$
  - $w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$
  - $w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$
  - $w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.00669 = -1.20669$
  - $w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.01122 = 1.08878$
  - $\theta_3 = \theta_3 + \Delta\theta_3 = 0.8 - 0.0038 = 0.7962$
  - $\theta_4 = \theta_4 + \Delta\theta_4 = -0.1 + 0.0015 = -0.0985$
  - $\theta_5 = \theta_5 + \Delta\theta_5 = 0.3 + 0.0127 = 0.3127$



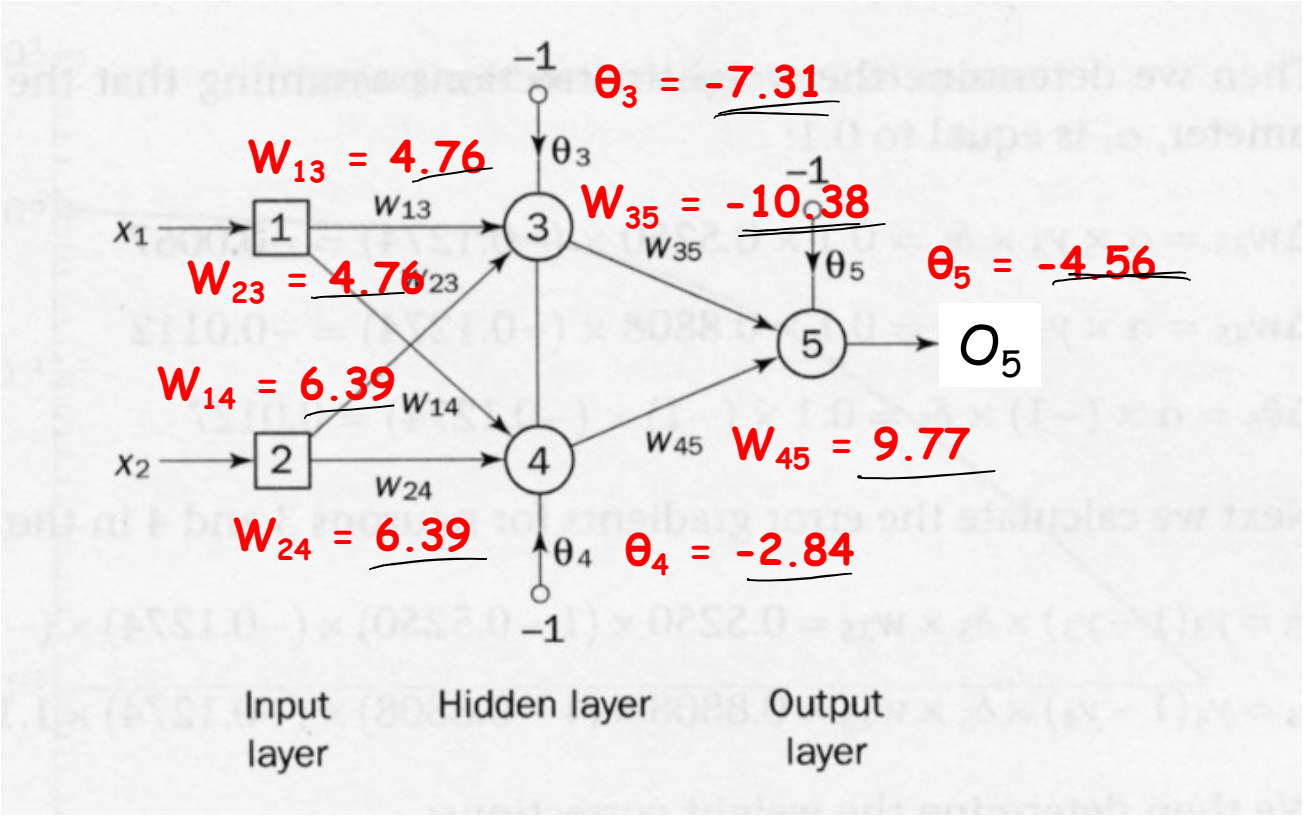
Step 4: Iterate through data

- after adjusting all the weights, repeat the forward pass and back pass for the next data point until all data points are examined
- repeat this entire exercise until the cost function is minimised
  - Eg. $C = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}(T^k{}_i \ln(O^k{}_i))$

where
n = nb of training examples
K = nb of classed

The Result…

- After 224 epochs, we get:
  - (1 epoch = going through all data once)



Error is minimized

1-0.9845

Inputs		Target Output	Actual Output
x_1	x_2	T	O
1	1	false (0)	0.0155
0	1	true (1)	0.9849
1	0	true (1)	0.9849
0	0	false (0)	0.0175

i	real distribution (false, true)	predicted distribution (false, true)
1	(1, 0)	(0.9845, 0.0155)
2	(0, 1)	(0.0151, 0.9849)
3	(0, 1)	(0.0151, 0.9849)
4	(1, 0)	(0.9825, 0.0175)

K=2 classes (false, true)

n= 4

$$C = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}(T^k{}_i \ln(O^k{}_i)) = \left(-\frac{1}{n}\right)($$

$$(1)ln(0.9845) + (0)ln(0.0155) \text{ // for i=1}$$

$$+ (0)ln(0.0151) + (1)ln(0.9849) \text{ // for i=2}$$

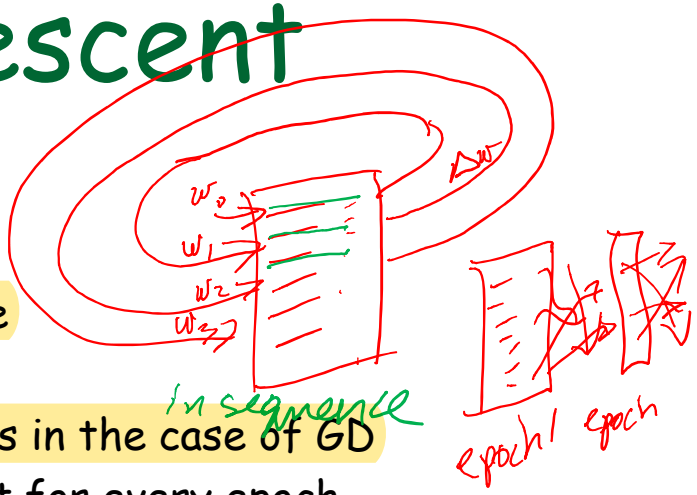
$$+ (0)ln(0.0151) + (1)ln(0.9849) \text{ // for i=3}$$

$$+ (1)ln(0.9825) + (0)ln(0.0175) \text{ // for i=4}$$

$$) = 0.01592 \quad < \varepsilon$$

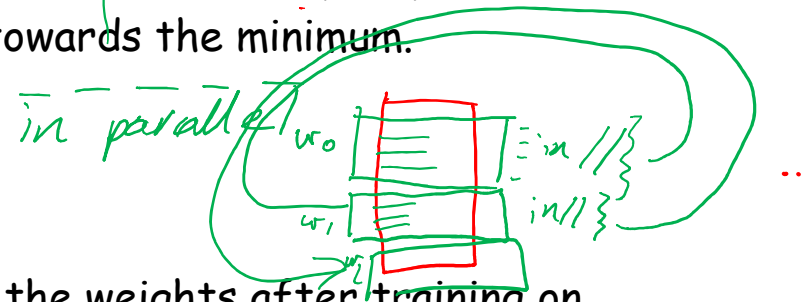
May be a local minimum…

Types of Gradient Descent



- Stochastic Gradient Descent (SGD)
  - updates the weights after each training example
  - often converges faster compared to GD
  - but the error function is not as well minimized as in the case of GD
  - to obtain better results, shuffle the training set for every epoch

- Batch Gradient Descent (GD)
  - updates the weights after 1 epoch
  - can be costly (time & memory) since we need to evaluate the whole training dataset before we take one step towards the minimum.



- MiniBatch Gradient Descent:
  - compromise between GD and SGD
  - cut your dataset into sections, and update the weights after training on each section

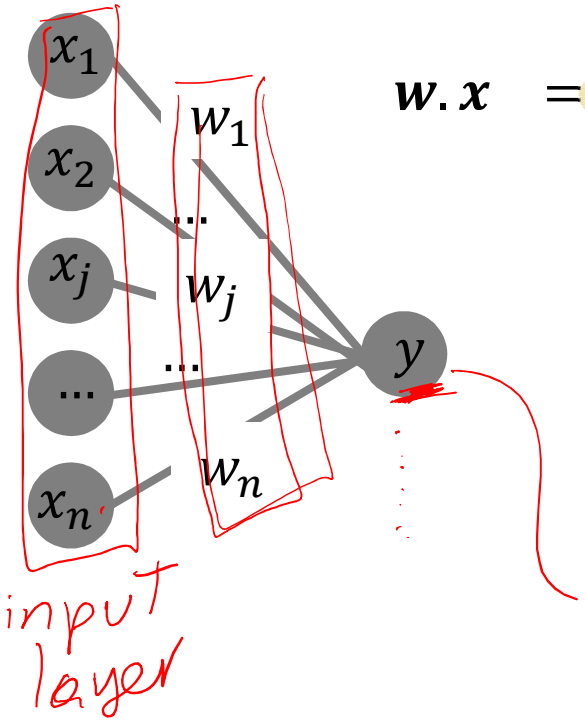
Remember your Linear Algebra

- Dot product (inner product) of 2 vectors

$$\boldsymbol{a}.\boldsymbol{b} = \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$$= \sum_{i=1}^{n} a_i b_i = (a_1 b_1 + a_2 b_2 + \cdots + a_n b_n)$$

so what?



$$\boldsymbol{w}.\boldsymbol{x} = [w_1 \quad w_2 \quad \dots \quad w_n] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$= \sum_{j=1}^{n} w_j x_j = (w_1 x_1 + w_2 x_2 + \cdots + w_n x_n)$$

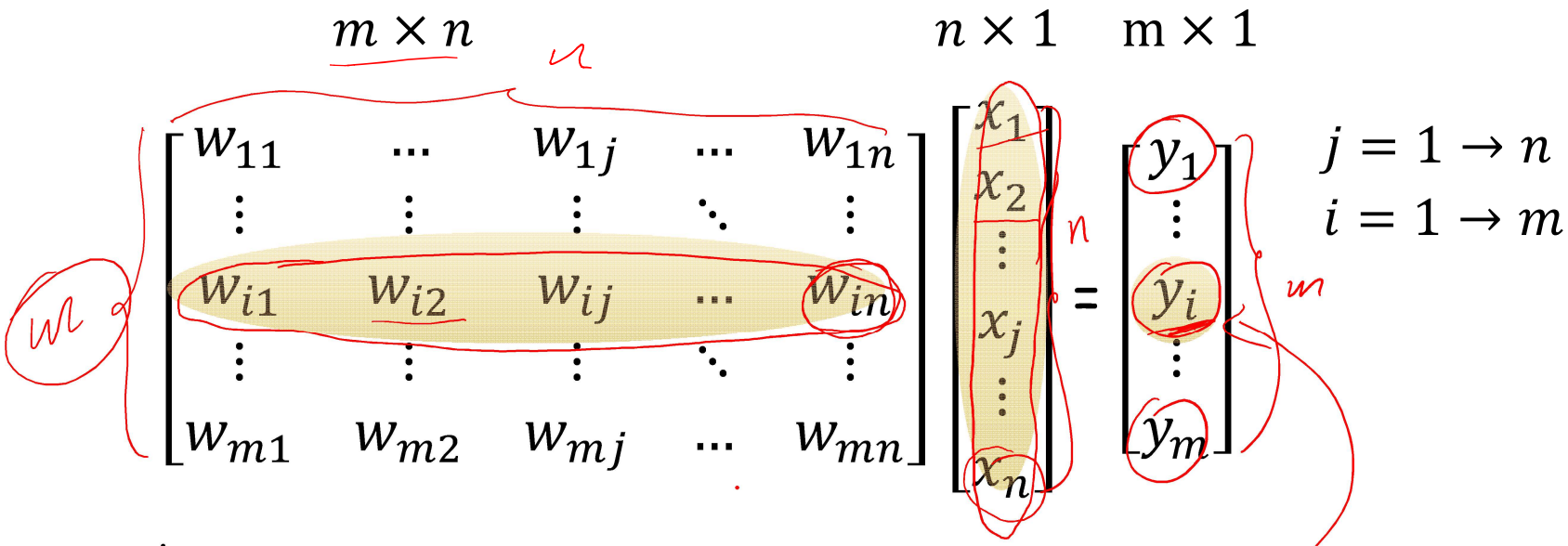
$= net$

$= y$

$sigmoid(net)$

Remember your Linear Algebra

- matrix-vector product

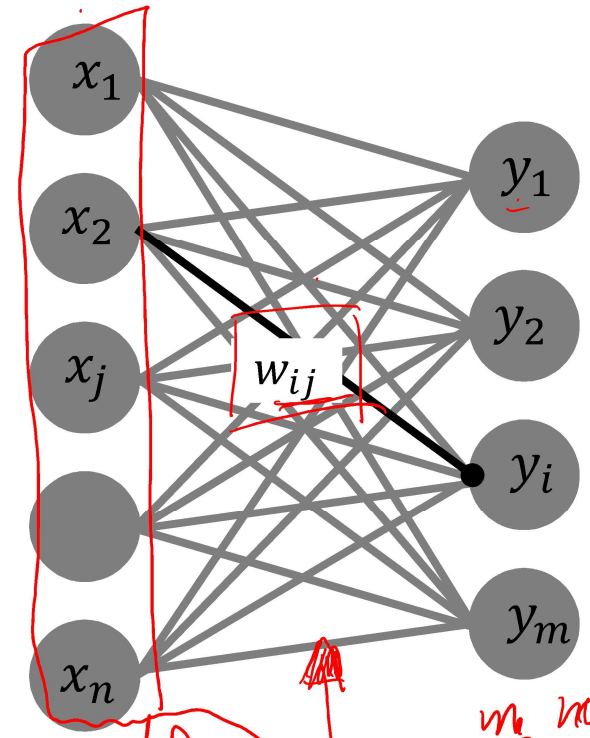


where:

$y_i = dot\ product\ of\ i^{th}\ row\ of\ \boldsymbol{W}\ with\ \boldsymbol{x}$

$$y_i = \sum_{j=1}^{n} w_{ij} x_j$$

so what?



m *hidden nodes*

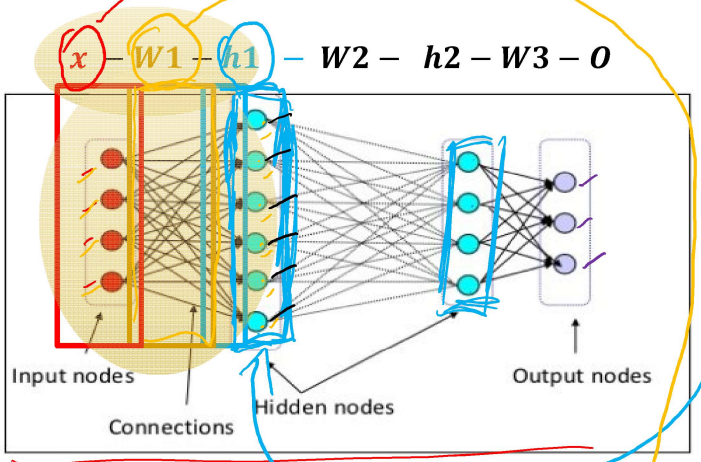
n *input nodes*

$w_{ij} = weight\ from\ node\ x_j\ to\ y_i$

$$y_i = sigmoid\left(\sum_{j=1}^{n} w_{ij}x_j\right)$$

$$\begin{bmatrix} w_{11} & \cdots & w_{1j} & \cdots & w_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & w_{ij} & \cdots & w_{in} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & w_{mj} & \cdots & w_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_i \\ \vdots \\ y_m \end{bmatrix}$$

Matrix Notation - Example

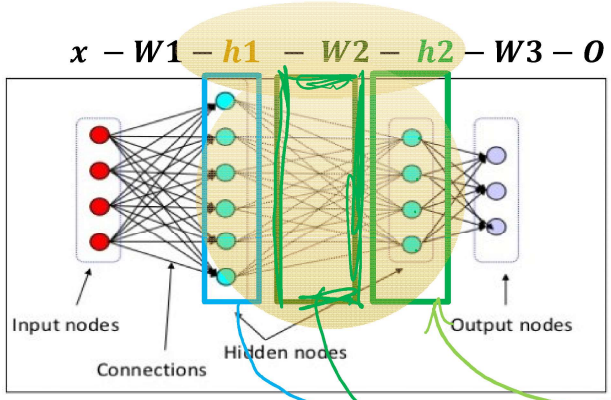


$$h1_i = sigmoid(net_{h1i}) = sigmoid\left(\sum_{j=1}^{4} W1_{ij}x_j\right)$$

$$h1_i = sigmoid\left(\underbrace{\begin{bmatrix} w1_{11} & w1_{12} & w1_{13} & w1_{14} \\ w1_{21} & w1_{22} & w1_{23} & w1_{24} \\ w1_{31} & w1_{32} & w1_{33} & w1_{34} \\ w1_{41} & w1_{42} & w1_{43} & w1_{44} \\ w1_{51} & w1_{52} & w1_{53} & w1_{54} \\ w1_{61} & w1_{62} & w1_{63} & w1_{64} \end{bmatrix}}_{6\times 4} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{4\times 1}\right) = \text{sigmoid}\begin{bmatrix} net_{h11} \\ net_{h12} \\ net_{h13} \\ net_{h14} \\ net_{h15} \\ net_{h16} \end{bmatrix} = \underbrace{\begin{bmatrix} h1_1 \\ h1_2 \\ h1_3 \\ h1_4 \\ h1_5 \\ h1_6 \end{bmatrix}}_{6\times 1}$$

output

Repeat on next level

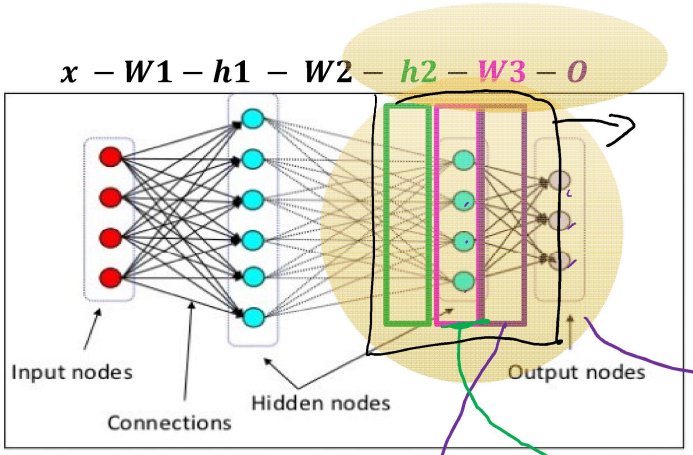


$$h2_i = sigmoid(net_{h2i}) = sigmoid\left(\sum_{j=1}^{6} w2_{ij} h1_j\right)$$

$$h2_i = sigmoid\left(\begin{bmatrix} w2_{11} & w2_{12} & w2_{13} & w2_{14} & w2_{15} & w2_{16} \\ w2_{21} & w2_{22} & w2_{23} & w2_{24} & w2_{25} & w2_{26} \\ w2_{31} & w2_{32} & w2_{33} & w2_{34} & w2_{35} & w2_{36} \\ w2_{41} & w2_{42} & w2_{43} & w2_{44} & w2_{45} & w2_{46} \end{bmatrix}_{4\times 6} \begin{bmatrix} h1_1 \\ h1_2 \\ h1_3 \\ h1_4 \\ h1_5 \\ h1_6 \end{bmatrix}_{6\times 1}\right) = \text{sigmoid}\left(\begin{bmatrix} net_{h21} \\ net_{h22} \\ net_{h23} \\ net_{h24} \end{bmatrix}\right) = \begin{bmatrix} h2_1 \\ h2_2 \\ h2_3 \\ h2_4 \end{bmatrix}_{4\times 1}$$

see previous slide

Repeat on next level



$$O_i = sigmoid(net_{Oi}) = sigmoid\left(\sum_{j=1}^{4} W3_{ij} h2_j\right)$$

$$O_i = sigmoid\left(\begin{bmatrix} w3_{11} & w2_{12} & w2_{13} & w2_{14} \\ w3_{21} & w2_{22} & w2_{23} & w2_{24} \\ w3_{31} & w2_{32} & w2_{33} & w2_{34} \end{bmatrix}_{3\times 4} \begin{bmatrix} h2_1 \\ h2_2 \\ h2_3 \\ h2_4 \end{bmatrix}_{4\times 1}\right) = \text{sigmoid}\left(\begin{bmatrix} net_{O1} \\ net_{O2} \\ net_{O3} \end{bmatrix}\right) = \begin{bmatrix} O_1 \\ O_2 \\ O_3 \end{bmatrix}_{3\times 1}$$

W_3

output h2

Neural Networks

- Disadvantage:
  - result is not easy to understand by humans (set of weights compared to decision tree)… it is a black box
- Advantage:
  - robust to noise in the input (small changes in input do not normally cause a change in output) and graceful degradation

Today

1. Introduction to ML ✓
2. Naïve Bayes Classification ✓
   a. Application to Spam Filtering ✓
3. Decision Trees ✓
4. (Evaluation ✓
5. Unsupervised Learning) ✓
6. Neural Networks ✓
   a. Perceptrons ✓
   b. Multi Layered Neural Networks ✓

Up Next

Part 4: Search

COMP 472: Artificial Intelligence

State Space Search part #3

State Space Representation video #1

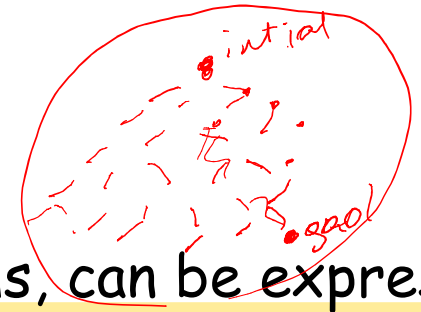
- Russell & Norvig – Sections 3.1-3.3

Today

video #1

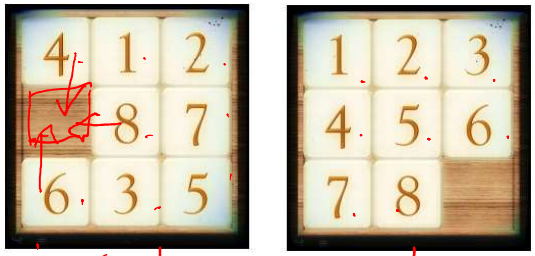
1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

Motivation



1970

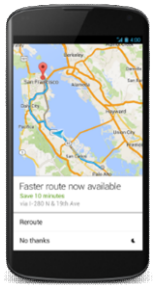
- Many AI problems, can be expressed in terms of going from an initial state to a goal state
  - *Ex: to solve a puzzle, to drive from home to Concordia…*



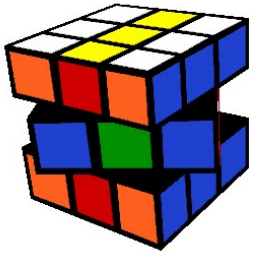
8-puzzle

15 moves

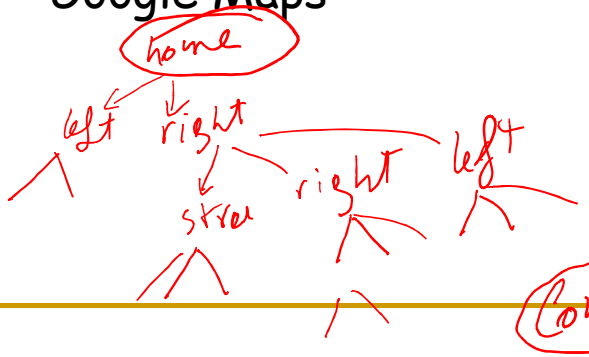
200 moves



Google Maps

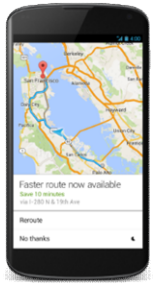


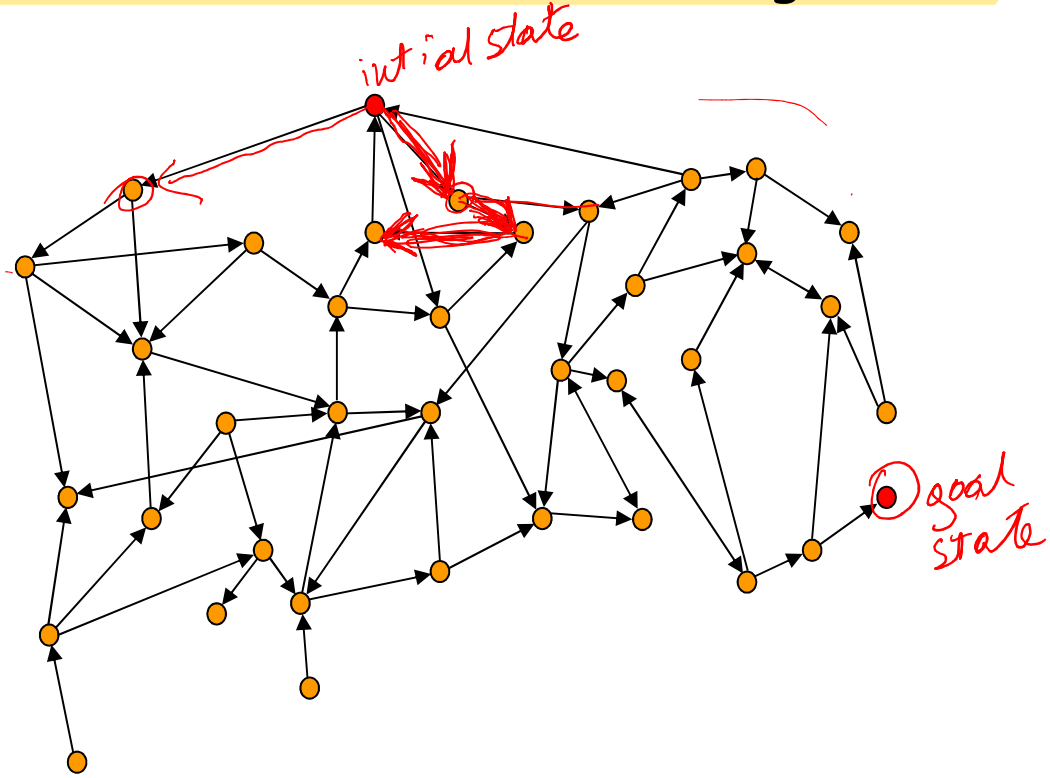
Rubik's cube



Motivation

- Often, there is no direct way to find a solution to go from the initial state to a goal state
- but we can list the possibilities and search through them





Example: 8-Puzzle

State: Any arrangement of 8 numbered tiles and an empty tile on a 3x3 board

8	2	
3	4	7
5	1	6

1	2	3
4	5	6
7	8	

Initial state → Goal state

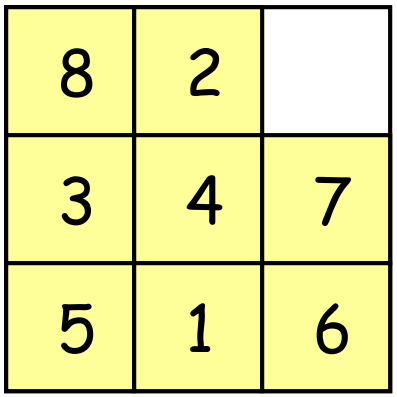
there are several standard goals states for the 8-puzzle

1	2	3
4	5	6
7	8	

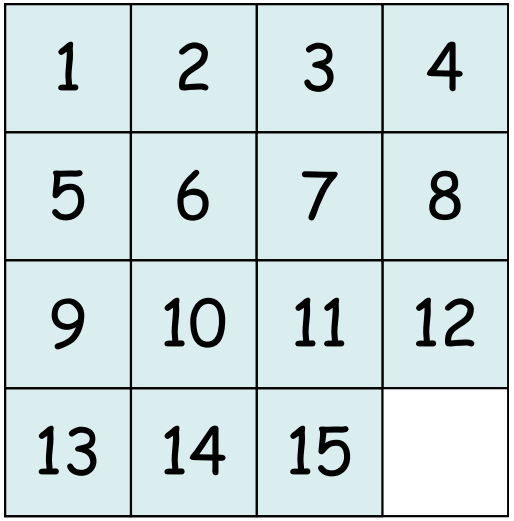
1	2	3
8		4
7	6	5

...

(n^2-1)-puzzle



8-puzzle



15-puzzle

. . . .

15-Puzzle

Invented in 1874 by Noyes Palmer Chapman …
but Sam Loyd claimed he invented it!

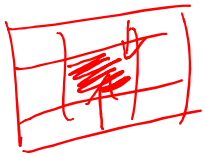
Sam Loyd

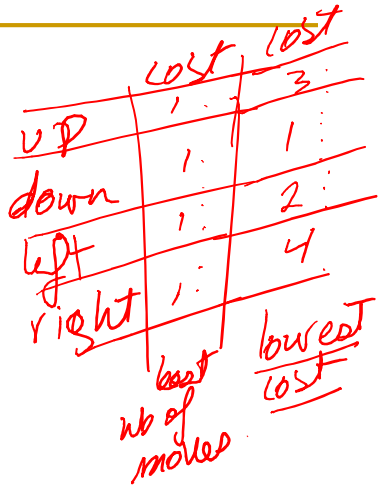
see Sam Loyd's book of puzzles:
https://archive.org/stream/CyclopediaOfPuzzlesLoyd/Cyclopedia_of_Puzzles_Loyd#mode/2up

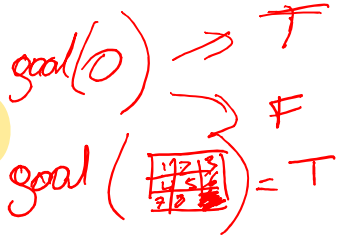
https://en.wikipedia.org/wiki/Sam_Loyd

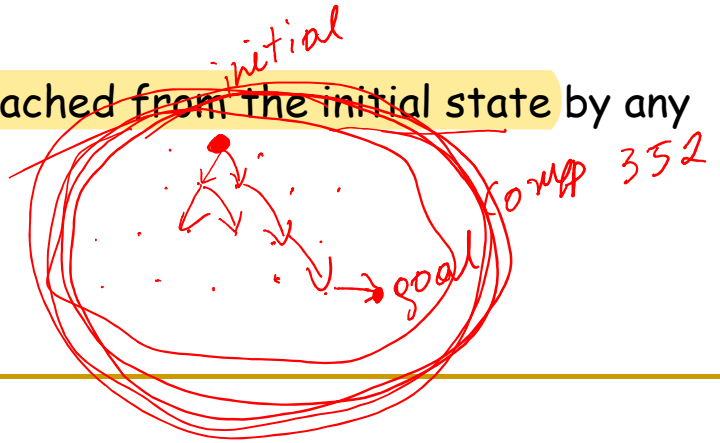
State Space

- Problem is represented by:
  1. Initial State
     - starting state
     - ex. unsolved puzzle, being at home
  2. Set of operators
     - actions responsible for transition between states
  3. Goal test function
     - Applied to a state to determine if it is a goal state
     - ex. solved puzzle, being at Concordia
  4. Path cost function
     - Assigns a cost to a path to tell if a path is preferable to another
- State space:
  - the set of all states that can be reached from the initial state by any sequence of action
- Search algorithm:
  - how the search space is visited

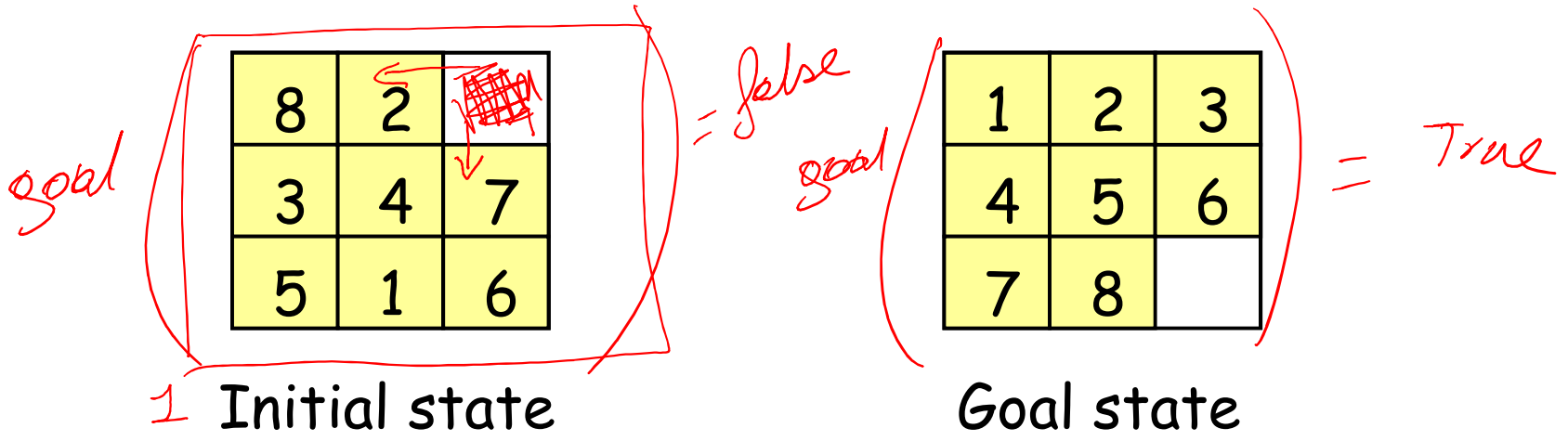








Example: The 8-puzzle



1 Initial state

Goal state

2 Set of operators:

blank moves up, blank moves down, blank moves left, blank moves right

3 Goal test function:

state matches the goal state

4 Path cost function:

each movement costs 1

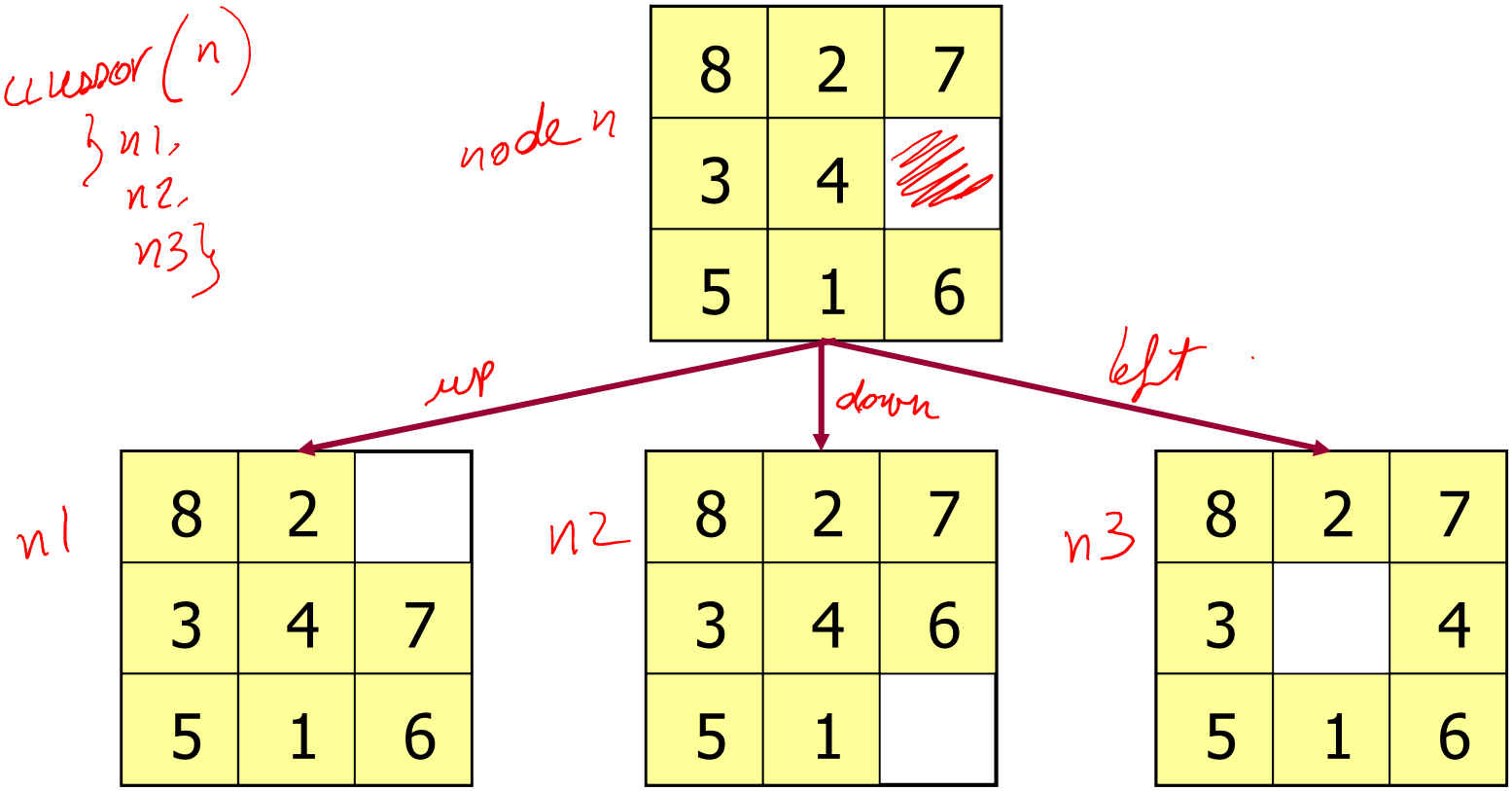
so the path cost is the length of the path (the number of moves)

source: G. Luger (2005)

8-Puzzle: Successor Function

children

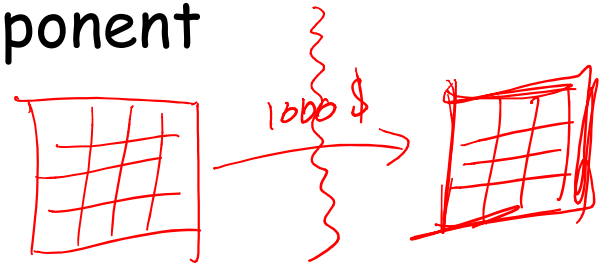
successor(n) = {n1, n2, n3}

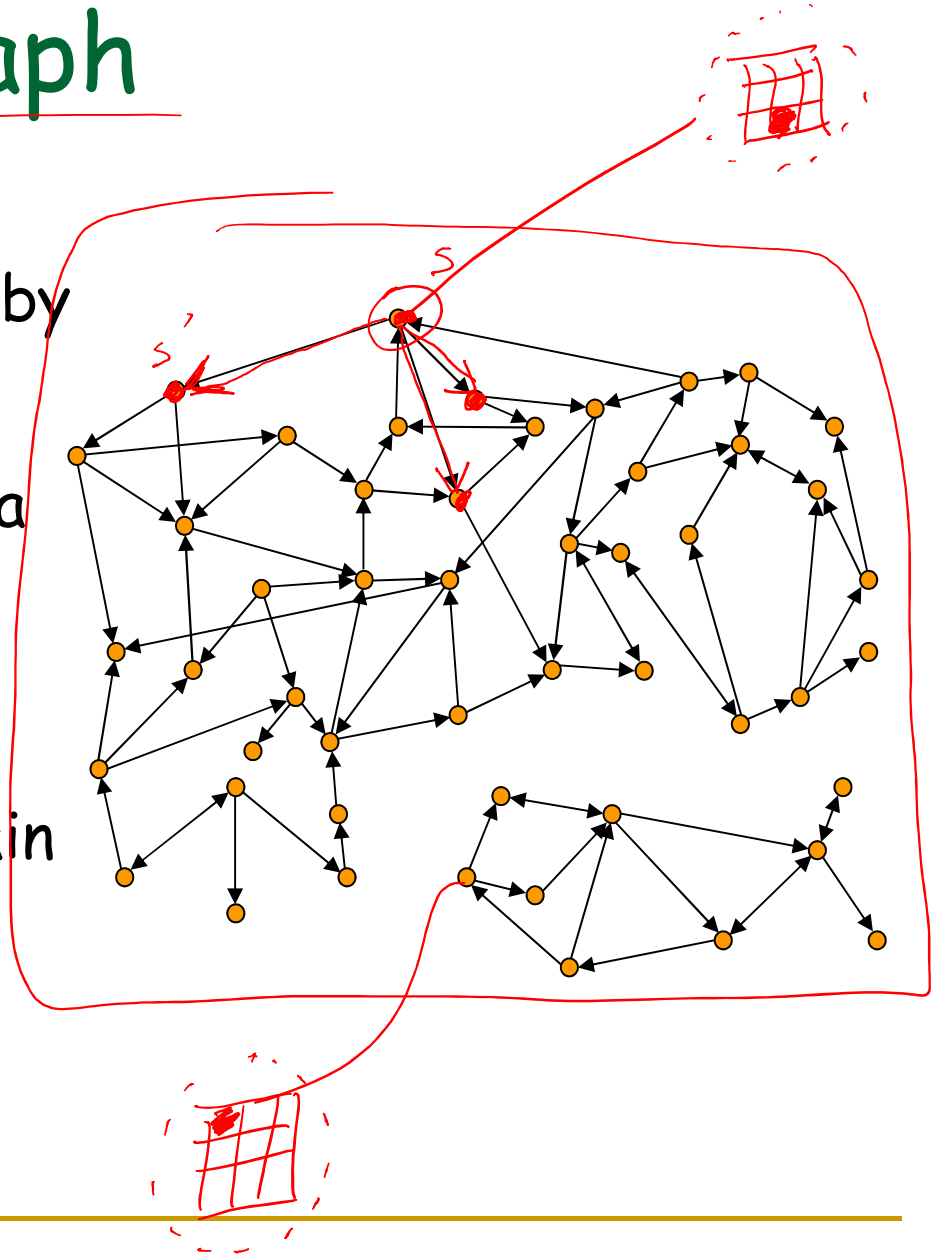


Search is about the exploration of alternatives

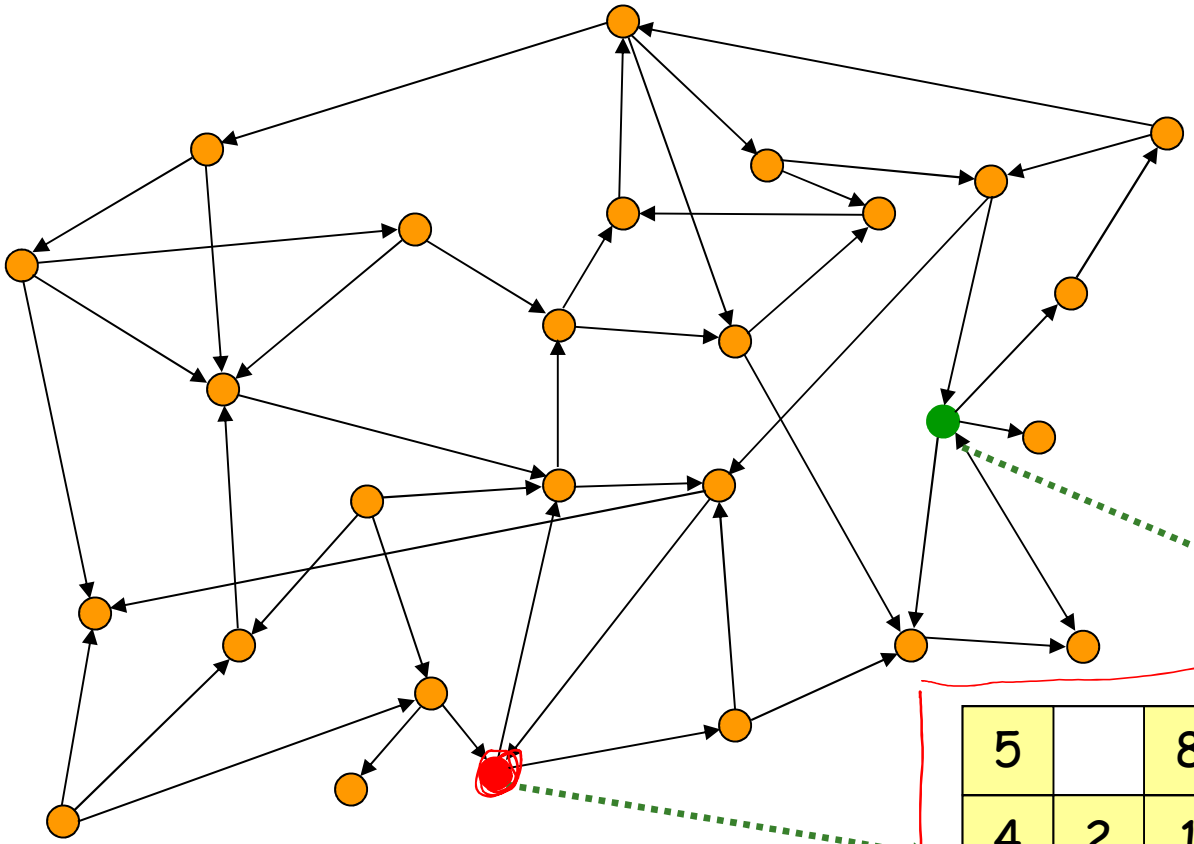
State Space Graph

- Each state is represented by a distinct node
- An arc (or edge) connects a node s to a node s' if $s' \in \text{SUCCESSOR}(s)$
- The state graph may contain more than one connected component





Just to make sure we're clear…



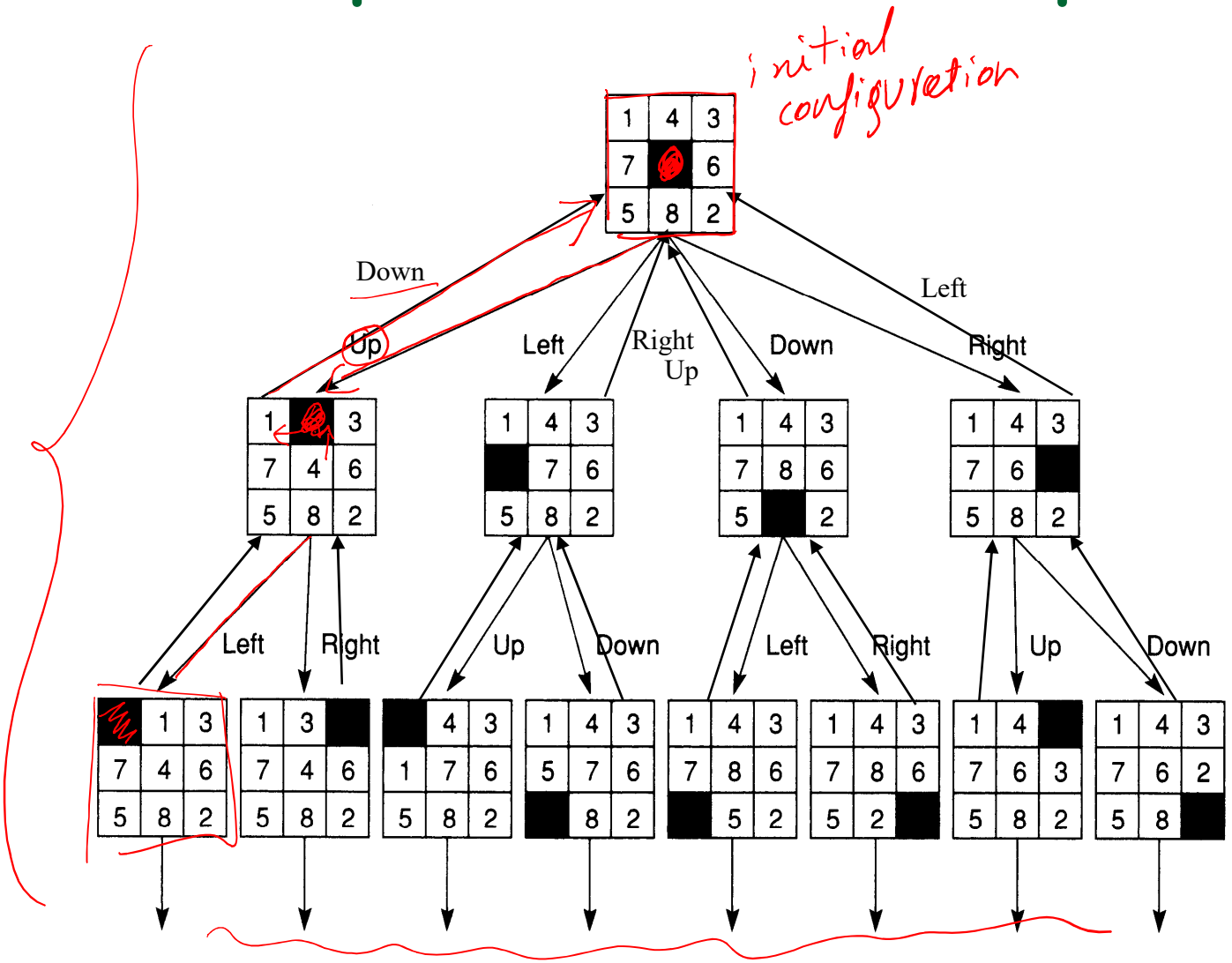
5		8
4	2	1
7	3	6

Initial state

1	2	3
4	5	6
7	8	

Goal state

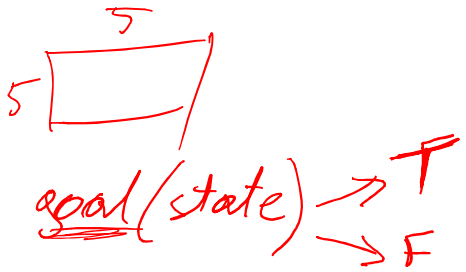
State Space for the 8-puzzle



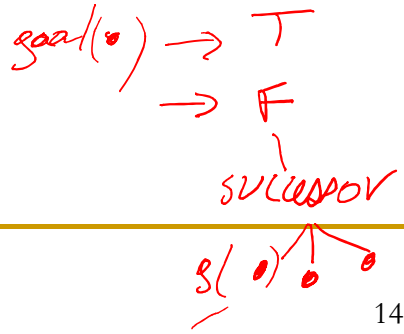
source: G. Luger (2005)

Size of state spaces

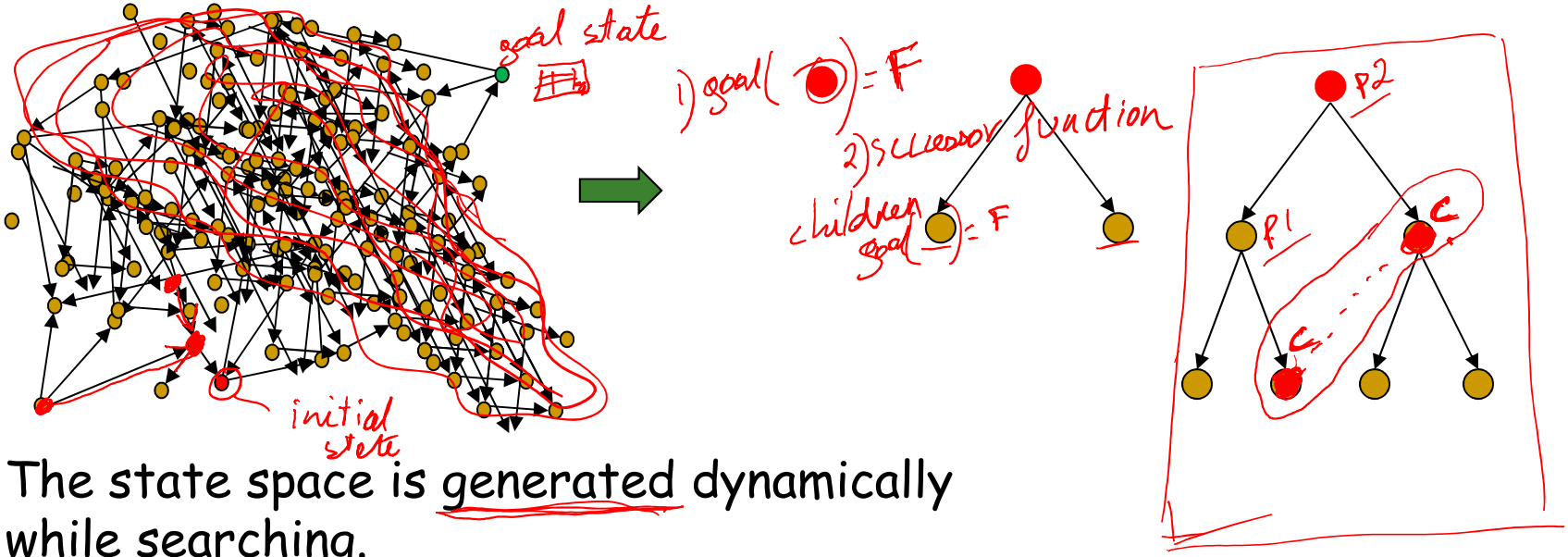
- For the (n-1)-puzzle:
  - Nb of states:
    - 8-puzzle --> 9! = 362,880 states
    - 15-puzzle --> 16! ~ 2.09×10^{13} states
    - 24-puzzle --> 25! ~ 10^{25} states
  - At 100 millions states/sec:
    - 8-puzzle --> 0.036 sec
    - 15-puzzle --> ~ 55 hours
    - 24-puzzle --> > 10^{9} years



- For real problems:
  - state spaces are way too large to be generated in advance and searched after
  - so it is generated dynamically while searching.



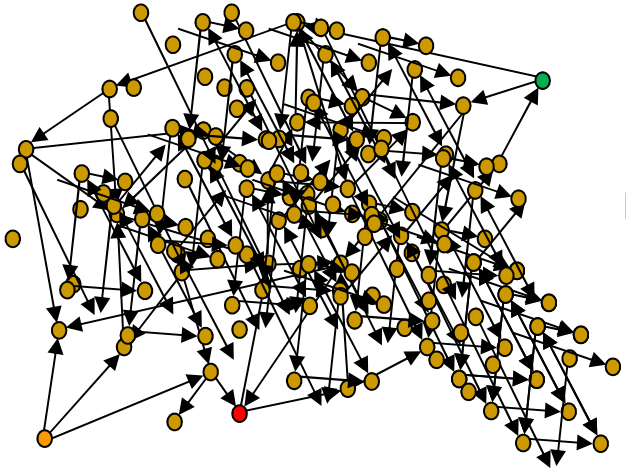
State Space Graph as a Search Tree



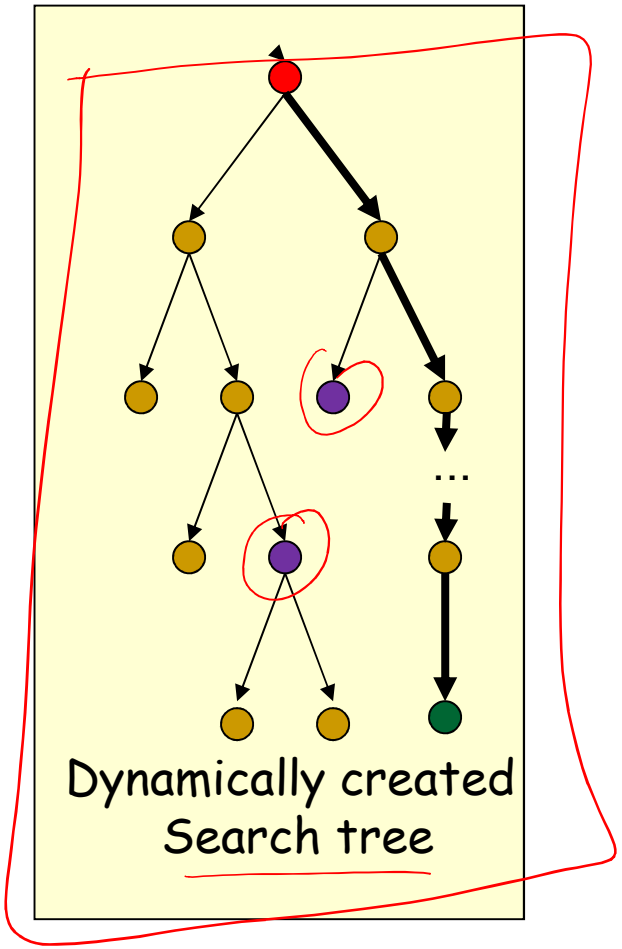
- The state space is generated dynamically while searching.
  - we explore a node:
    - if it the goal, stop and trace back the path from the initial node / solution path
    - if it is not the goal, then generating its successors/children and explore these recursively
  - to avoid cycles, the search algorithm will check for duplicate nodes.

Search tree

State Space Graph as a Search Tree



Theoretical State Space Graph



So now, we just need an efficient search algorithm

Today

1. State Space Representation 
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

Up Next

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

video #2

COMP 352

novelty

COMP 472 Artificial Intelligence
State Space Search part #3
Uninformed Search video #2

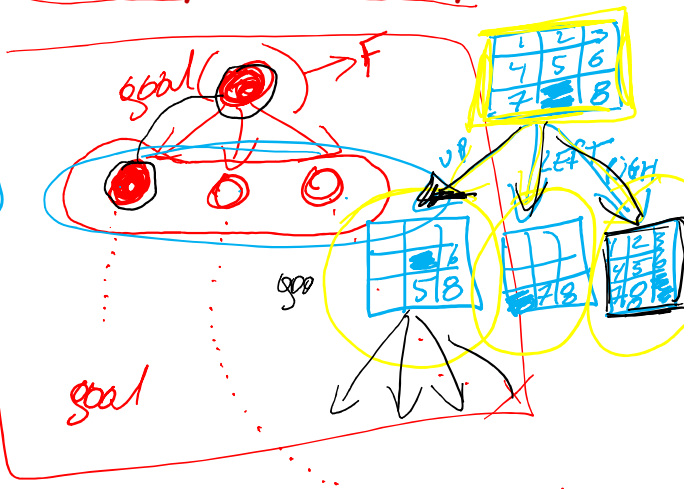
- Russell & Norvig – Section 3.4
- see also: https://www.javatpoint.com/ai-uninformed-search-algorithms

Today

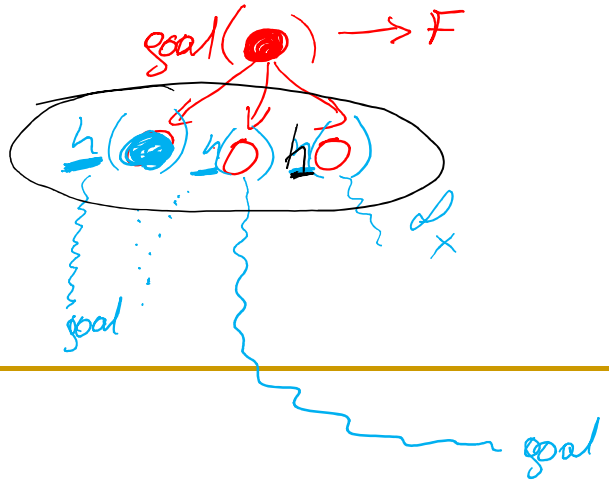
1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Best-First
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

Uninformed VS Informed Search

- Uninformed search
  - all nodes are equally promising, so we explore them systematically
  - aka: systematic/blind/brute force search
  - many algorithms:
    1. Breadth-first search
    2. Depth-first seach
    3. Uniform-cost search
    4. Depth-limited search
    5. Iterative deepening search
    6. …



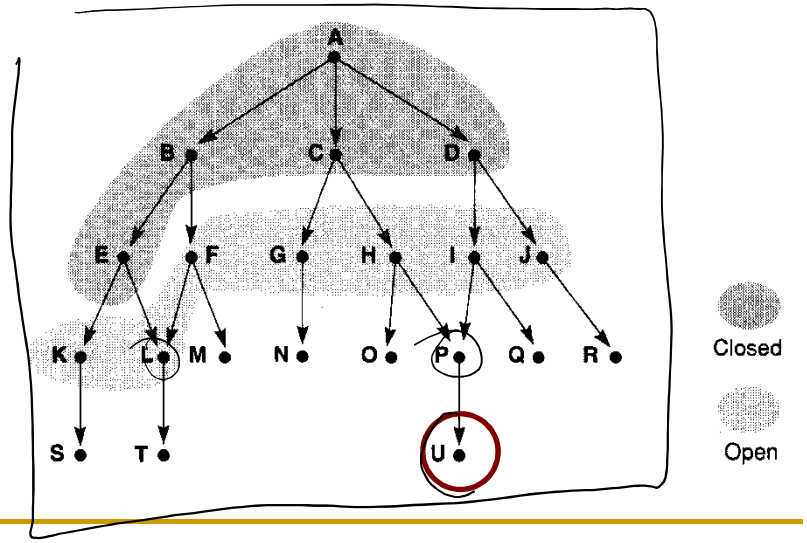
- Informed search (heuristic search)
  - we try to identify which nodes seem more promising, and explore these first
  - many algorithms:
    1. Hill climbing
    2. Gready Best-First search
    3. Algorithms A and A*
    4. …



Data Structures

- Most search strategies require:
  - ***open list*** (aka the frontier) To-DO
    - lists generated nodes not yet expanded
    - order of nodes controls order of search
  - ***closed list*** (aka the explored set)
    - stores all the nodes that have already been visited (to avoid cycles).
- ex:

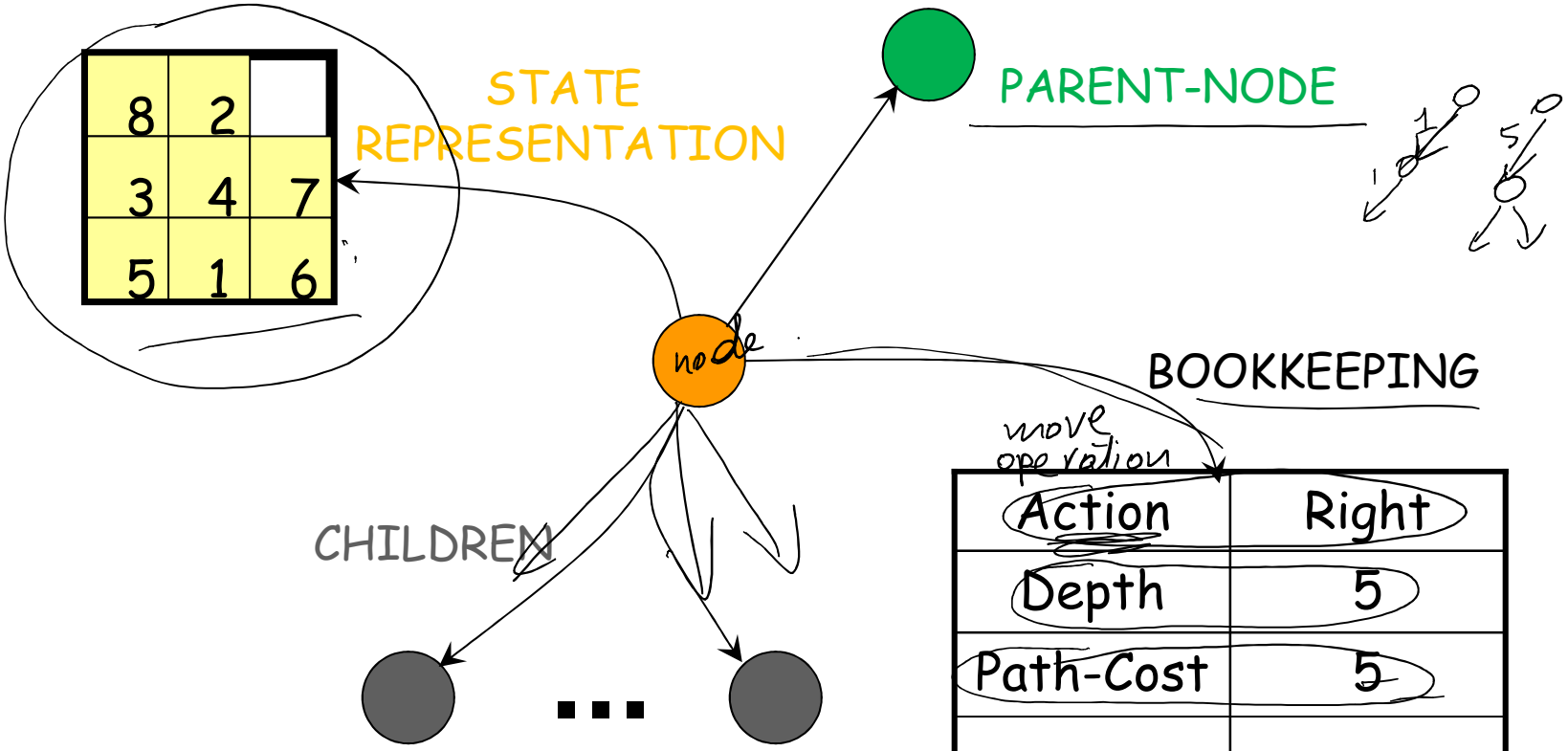
Closed = [A, B, C, D, E]
Open = [F, G, H, I, J, K, L]



source: G. Luger (2005)

Data Structures

- state space representation: To trace back the solution path after the search, each node in the lists contain:



Depth = length of path from root to node

robotics.stanford.edu/~latombe/cs121/2003/home.htm

Generic Search Algorithm

1. Initialize the open list with the initial node s_0 (top node)
2. Initialize the closed list to empty
3. Repeat
   a) If the open list is empty, then exit with failure.
   b) Else, take the first node s from the open list.
   c) If s is a goal state, exit with success. Extract the solution path from s to s_0
   d) Else, insert s in the closed list (s has been visited /expanded)
   e) Insert the successors of s in the open list in a certain order if they are not already in the closed and/or open lists (to avoid cycles)

Notes:

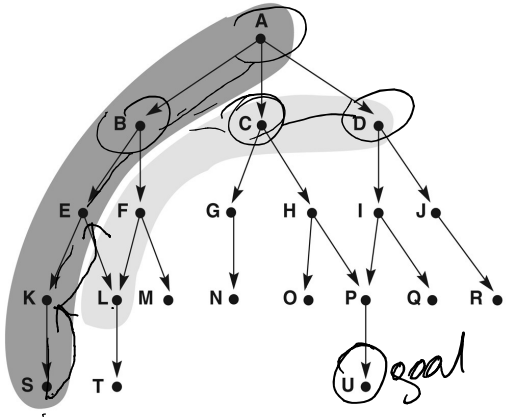
- The order of the nodes in the open list depends on the search strategy

Today

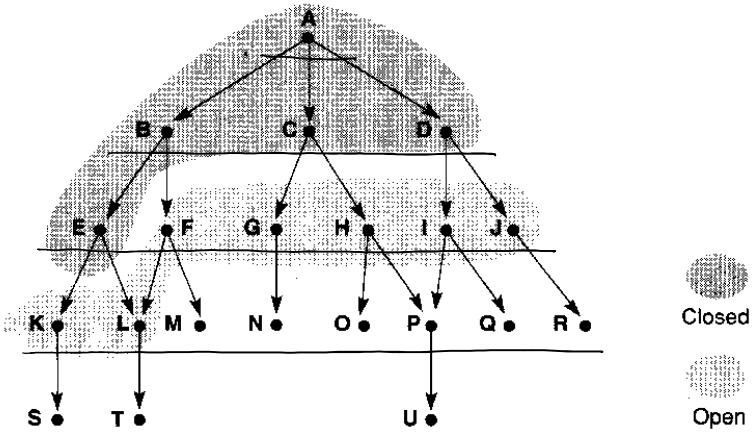
1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first Search and Depth-first Search
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Best-First
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

Depth-first vs Breadth-first Search

- Depth-first (DFS):
  - visit successors before siblings
  - Open list is a stack



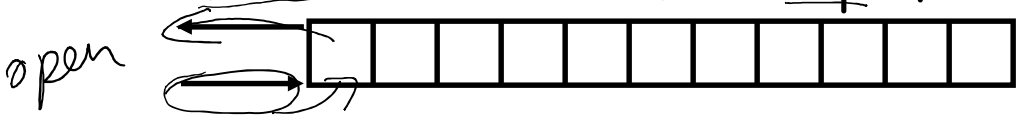
- Breadth-first (BFS):
  - visit siblings before successors
  - ie. visit level-by-level
  - open list is a queue



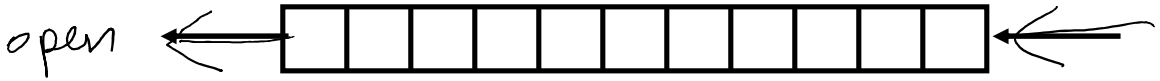
source: G. Luger (2005)

DFS and BFS

- DFS and BFS differ only in the way they order nodes in the open list:
  - DFS uses a stack:
    - nodes are added on the top of the list.



  - BFS uses a queue:
    - nodes are added at the end of the list.



Breadth-First Search

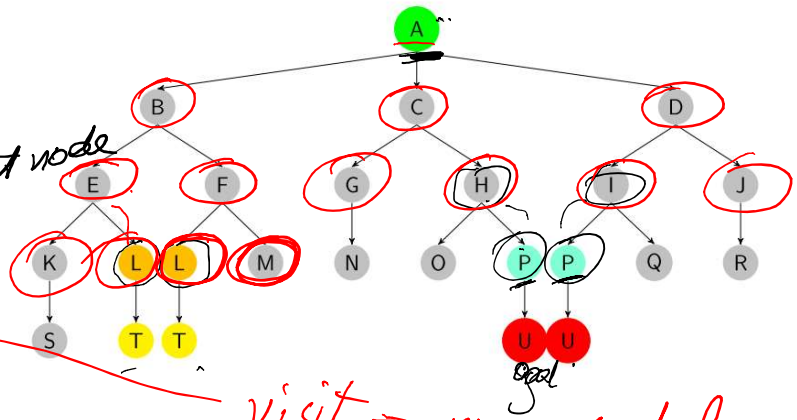
```
begin
   open := [Start];                                          % initialize
   closed := [ ];
   while open ≠ [ ] do                                       % states remain
      begin
         remove leftmost state from open, call it X;
            if X is a goal then return SUCCESS               % goal found
               else begin
                  generate children of X;
                  put X on closed;
                  discard children of X if already on open or closed;   % loop check
                  put remaining children on right end of open           % queue
               end
      end
   return FAIL                                               % no states left
end.
```

source: G. Luger (2005)

Breadth-First Search Example

- BFS: (open is a queue)

Assume U is goal state



name of node
name of parent node

visit ⇒ run goal function

1. open = [A-null] closed = []
2. open = [B-A C-A D-A] closed [A]
3. open = [C-A D-A E-B F-B] closed = [B A]
4. open = [D-A E-B F-B G-C H-C] closed = [C B A]
5. open = [E-B F-B G-C H-C I-D J-D] closed = [D C B A]
6. open = [F-B G-C H-C I-D J-D K-E L-E] closed = [E D C B A]
7. open = [G-C H-C I-D J-D K-E L-E M-F] *as L is already in open* closed = [F E D C B A]
8. and so on until either U is found or open = []

search path = A B C D E F G H I J K L M N ... U // closed list

solution path = A C H P U with a cost of 4 // extract by following the parent pointer

source: G. Luger (2005)

Depth-First Search

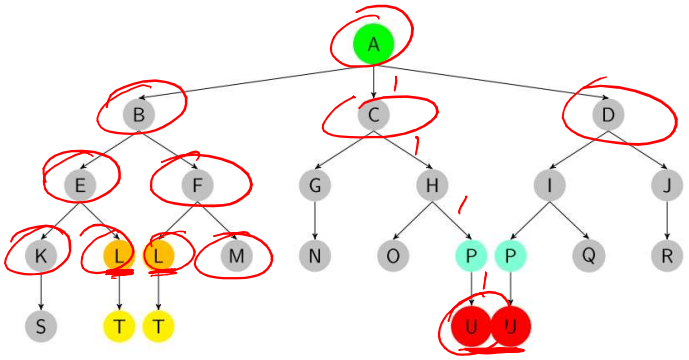
```
begin
   open := [Start];                                          % initialize
   closed := [ ];
   while open ≠ [ ] do                                       % states remain
      begin
         remove leftmost state from open, call it X;
         if X is a goal then return SUCCESS                  % goal found
            else begin
               generate children of X;
               put X on closed;
               discard children of X if already on open or closed;   % loop check
               put remaining children on left end of open          % stack
            end
      end;
   return FAIL                                               % no states left
end.
```

source: G. Luger (2005)

Depth-First Search Example

- DFS: (open is a stack)

Assume U is goal state



1. open = [A-null] closed = []
2. open = [B-A C-A D-A] closed [A]
3. open = [E-B F-B C-A D-A] closed = [B A]
4. open = [K-E L-E F-B C-A D-A] closed = [E B A]
5. open = [S-K L-E F-B C-A D-A] closed = [K E B A]
6. open = [L-E F-B C-A D-A] closed = [S K E B A]
7. open = [T-L F-B C-A D-A] closed = [L S K E B A]
8. open = [F-B C-A D-A] closed = [T L S K E B A]
9. open = [M-F C-A D-A] *as L is already on closed* closed = [F T L S K E B A]
10. open = [C-A D-A] closed = [M F T L S K E B A]
11. open = [G-C H-C D-A] closed = [C M F T L S K E B A]

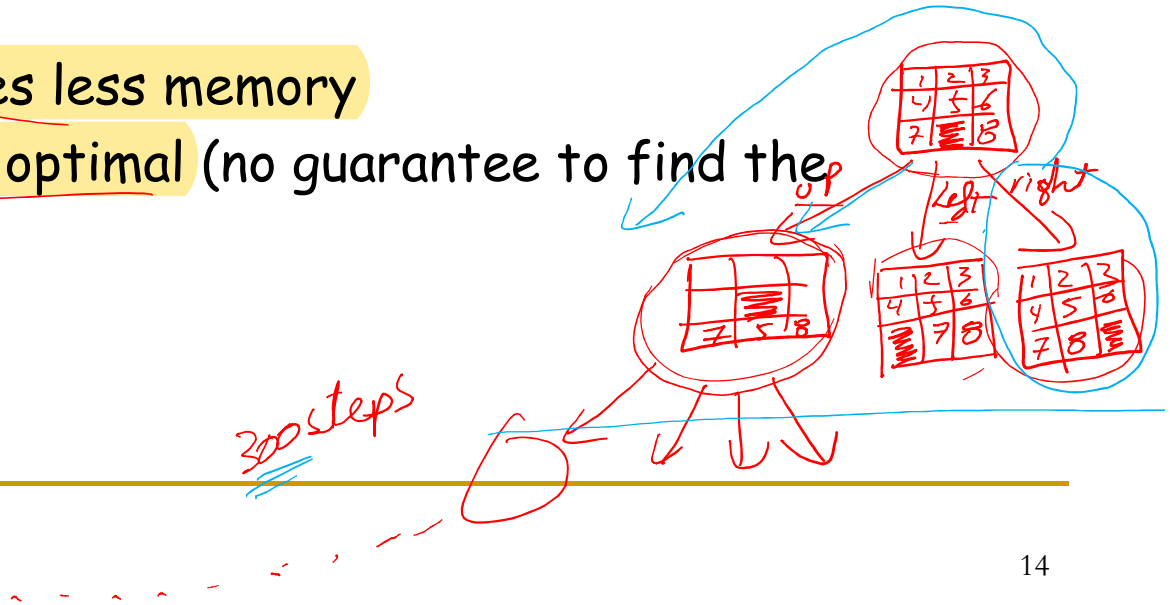
search path = A B E K S L .... U // closed list

solution path = A C H P U with a cost of 4

source: G. Luger (2005)

Depth-first vs. Breadth-first

- Breadth-first:
  - advantage: optimal, i.e. will always find shortest path
  - disadvantage:
    - high memory requirement as we need to keep all states of a level before expanding to the next level
    - exponential space for states required B^n // B=branching factor, n =level
- Depth-first:
  - advantage: Requires less memory
  - disadvantage: Not optimal (no guarantee to find the shortest path)



Today

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first Search and Depth-first Search
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Best-First
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

Depth-Limited Search

- Compromise for DFS :
  - Do depth-first but
  - with depth cutoff k (depth at which nodes are not expanded)

- Three possible outcomes:
  - Solution within your k limit :)
  - Failure (no solution)
  - Cutoff (no solution found within cutoff) X

- advantage: memory efficient – it's a DFS
- disadvantage: may not find a solution if it is below the cutoff

Today

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first search and Depth-first search
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Best-First
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

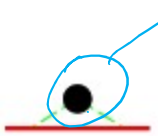
Iterative Deepening

- Combination of BFS and DFS:
  - do depth-first search, but
  - with a maximum depth before going to next level
  - i.e. Repeats depth first search with gradually increasing depth limits
- advantage:
  - Requires little memory (fundamentally, it's a depth first)
  - optimal: will find the shortest path (limited depth)
- disadvantage:
  - repeated traversal of the tree top

Iterative Deepening: Example

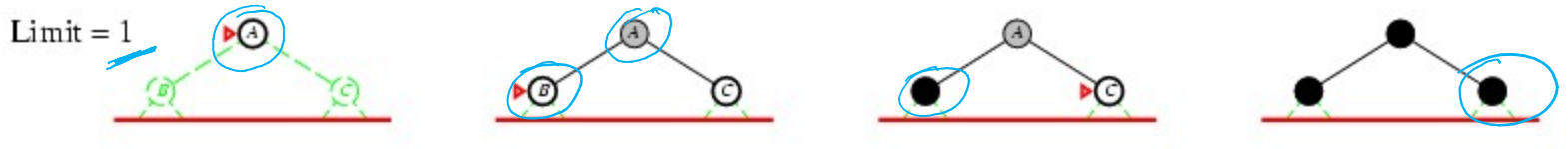
black node
⇒ node has been visited

Limit = 0



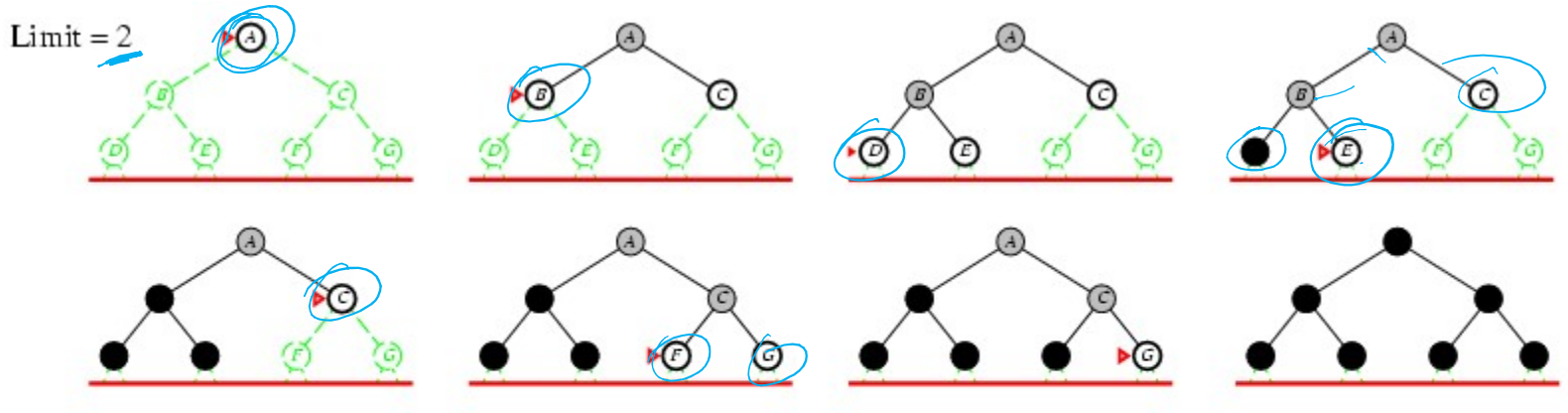
source: Russel & Norvig (2003)

Iterative Deepening: Example



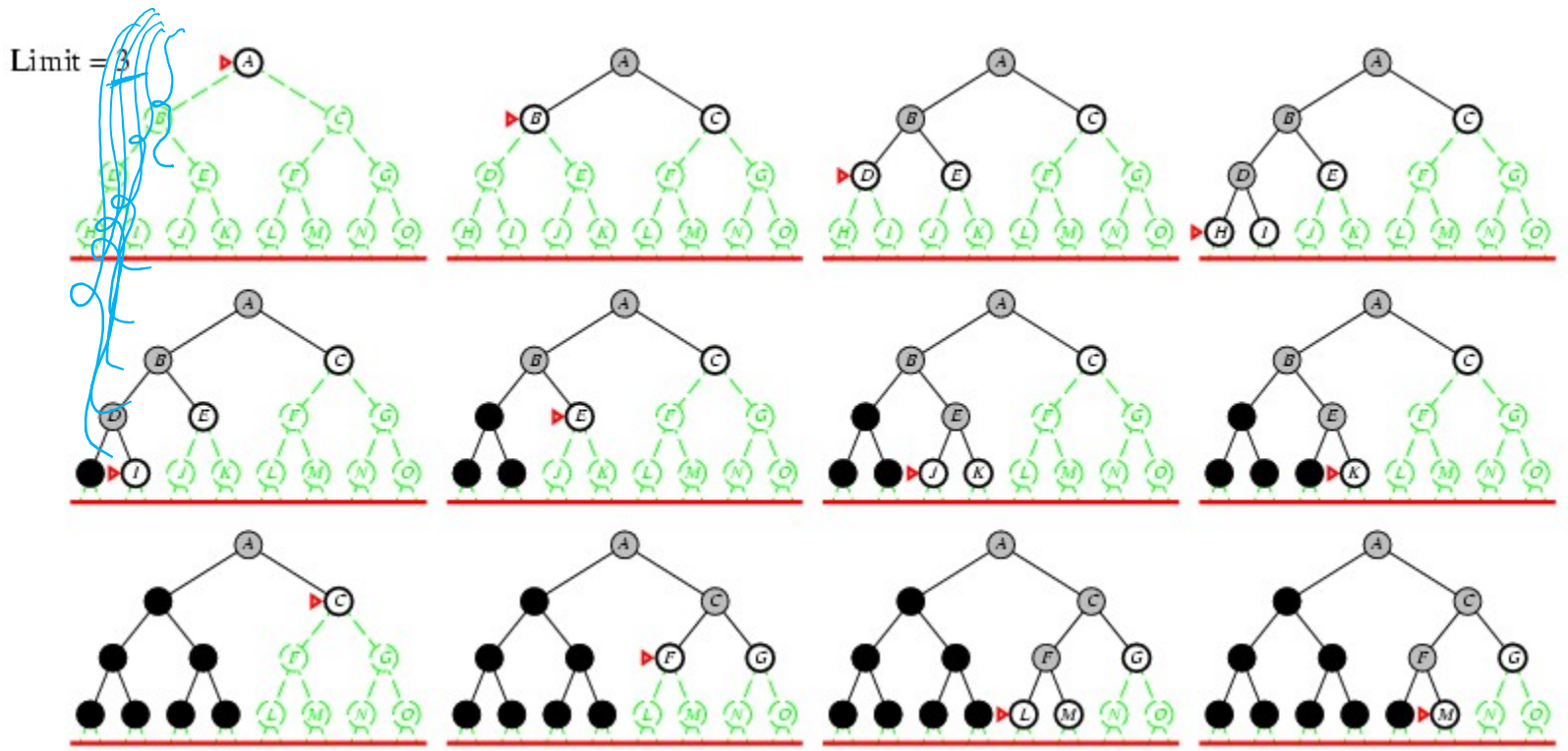
source: Russel & Norvig (2003)

Iterative Deepening: Example



source: Russel & Norvig (2003)

Iterative Deepening: Example



source: Russel & Norvig (2003)

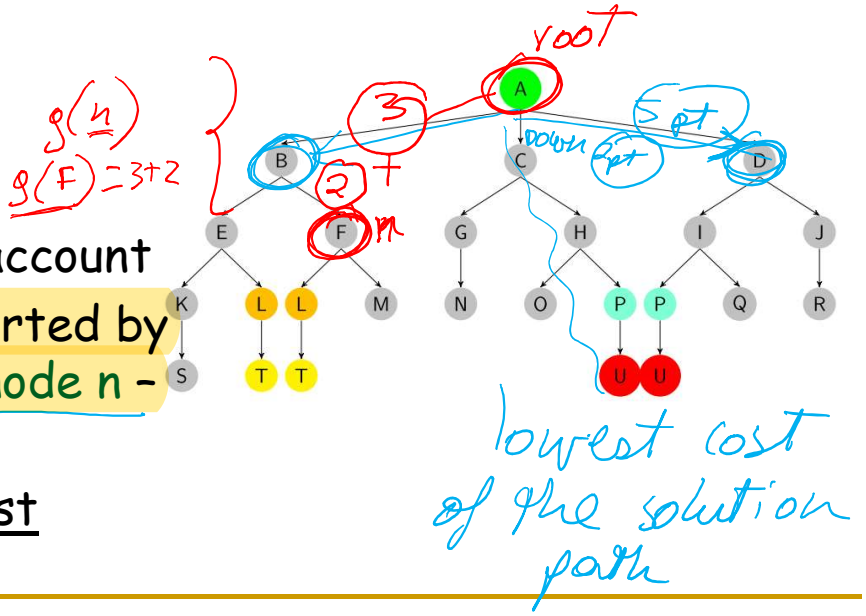
Today

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost

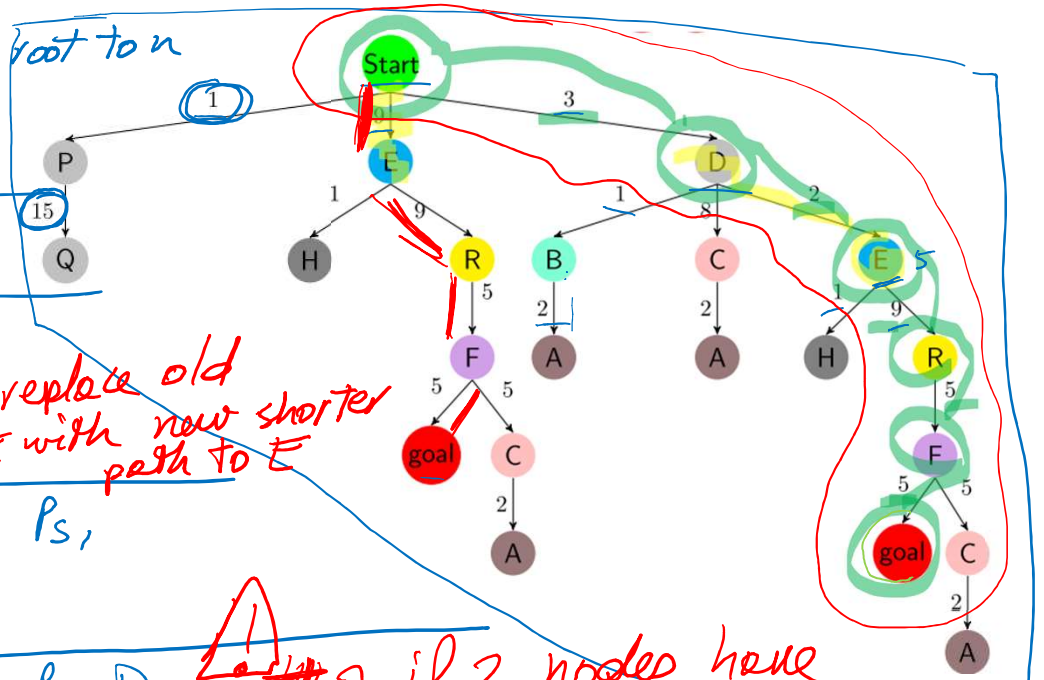
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Best-First
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

Uniform Cost Search

- all algorithms so far assume that all edges have the same cost
- but what if they have different costs ?
  - eg: move UP -> 2pts but move DOWN -> 1 pt
  - eg: cost(residential road) > cost(commercial road)
- Breadth First Search
  - uses OPEN as a priority queue sorted by the depth of nodes
  - guarantees to find the shortest solution path
- Uniform Cost Search
  - takes the cost of the edge into account
  - uses OPEN as a priority queue sorted by the total cost from the root to node n - *later we will call this g(n)*
  - guarantees to find the lowest cost solution path



Example



node
parent node
total cost from root to n

open | closed

S_{null} 0 | ∅

P_{S1} E_{S9} D_{S3}

P_{S1} D_{S3} E_{S9}

S_{null} 0 P_{S1}

Q_{P16} D_{S3} E_{S9}

D_{S3} E_{S9} Q_{P16}

S_{null} 0 P_{S1} D_{S3}

B_{D4} C_{D11} E_{D5} Q_{P16}

B_{D4} E_{D5} C_{D11} Q_{P16}

S_{null} 0 P_{S1} D_{S3} B_{D4}

A_{B6} E_{D5} C_{D11} Q_{P16}

E_{D5} A_{B6} C_{D11} Q_{P16}

E_{D5}

H_{E6} R_{E14} A_{B6} C_{D11} Q_{P16}

H_{E6} A_{B6} C_{D11} R_{E14} Q_{P16}

I skipped the last steps

#1 replace old E with new shorter path to E

#2 if 2 nodes have the same cost in the open list ⇒ arbitrary choice

solution path : Goal F R E D S cost of 24

Today

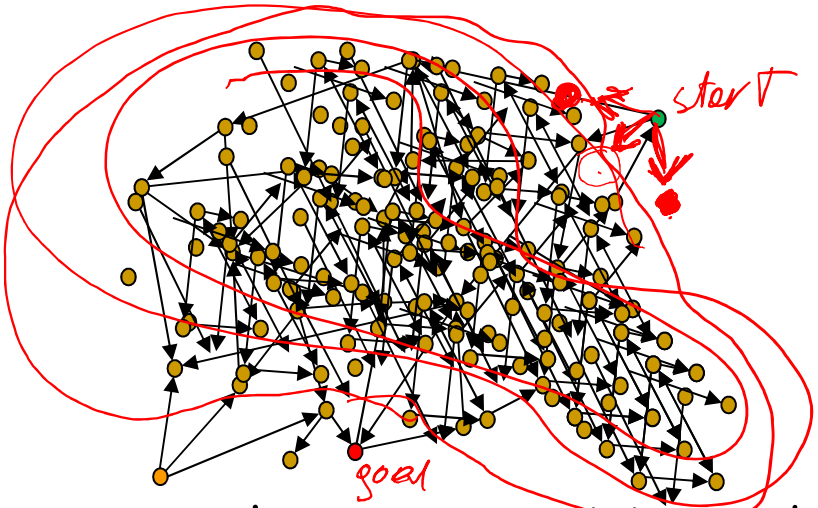
1. State Space Representation ✓
2. State Space Search ✓
   a) Overview ✓
   b) Uninformed search ✓
      1. Breadth-first and Depth-first ✓
      2. Depth-limited Search ✓
      3. Iterative Deepening ✓
      4. Uniform Cost ✓

      } 352

   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Best-First
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

Problem with Uninformed Search

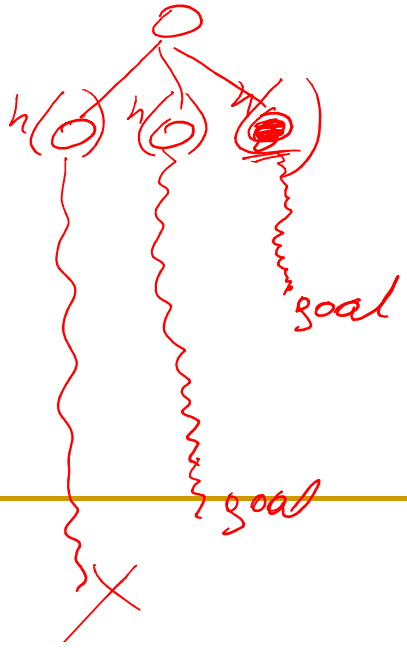
- inefficient for most AI problems, the state space is too large!
  - e.g. state space of all possible moves in chess = 10^{120}
  - 10^{75} = nb of molecules in the universe
  - 10^{26} = nb of nanoseconds since the "big bang"



- we need a way to try the most promising nodes first

Up Next

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) **Informed search**
      1. Intro to Heuristics
      2. Hill climbing
      3. Best-First
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary



COMP 472 Artificial Intelligence
State Space Search part #3
Intro to Heuristics video #3

- Russell & Norvig – Sections 3.5.1, 3.5.2, 4.1.1

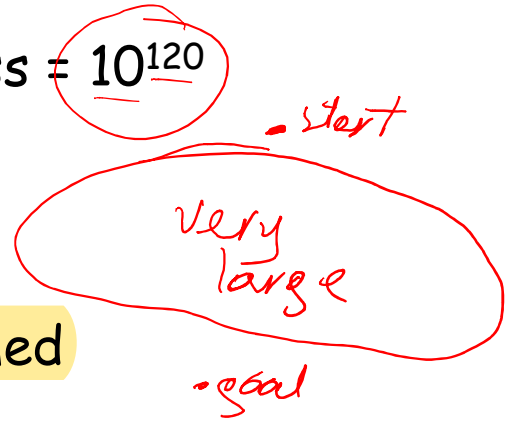
Today

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first Search and Depth-first Search
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

h(n)

Informed Search (aka heuristic search)

- Most of the time, it is not feasible to do a systematic search, the search space is too large
  - e.g. state space of all possible moves in chess = 10^{120}
    - 10^{75} = nb of molecules in the universe
    - 10^{26} = nb of nanoseconds since the "big bang"

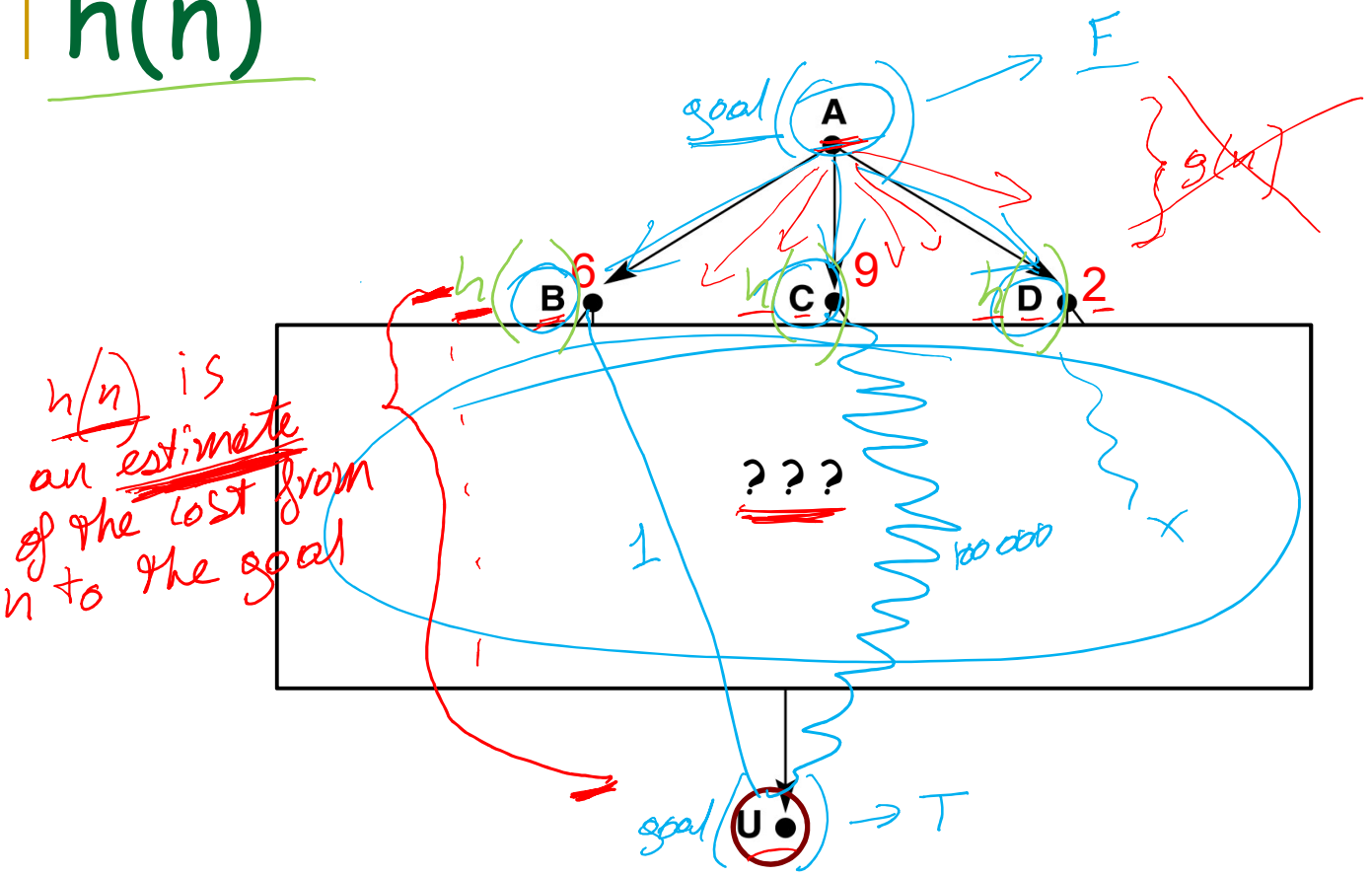


- so far, all search algorithms have been uninformed
- ie. all nodes are equally promising
- we need a way to visit the most promising nodes first
  - most-promising = close to the goal state
  - so we need a estimate function (i.e. a heuristic function h(n))
  - so the search is now called informed/heuristic search

Heuristic - *Heureka!*

- Heuristic search:
  - A technique that improves the efficiency of search
  - Focus on nodes that seem most promising according to some function $h(n)$
  - Need an evaluation function (heuristic function) to estimate ~~how close~~ of the cost a node is to the goal
- Heuristic function *h(n)*:
  - a rule of thumb, a good bet, an estimate
  - but has no guarantee to be correct
  - an approximation of the lowest cost from node n to the goal

h(n)



- **h(n)** = estimate of the lowest cost from *n* to *goal*

Designing Heuristics

- h(n) are highly dependent on the search domain
- A h(n) whose value is closer to the actual cost to the goal will lead to:
  - a shorter search path
  - less backtracking
  - i.e. less nodes are visited/searched for nothing
  - but this is not always the best idea...
    - it depends on the computational cost of h(n)

$h_1(n)$ // good estimate but quick to run

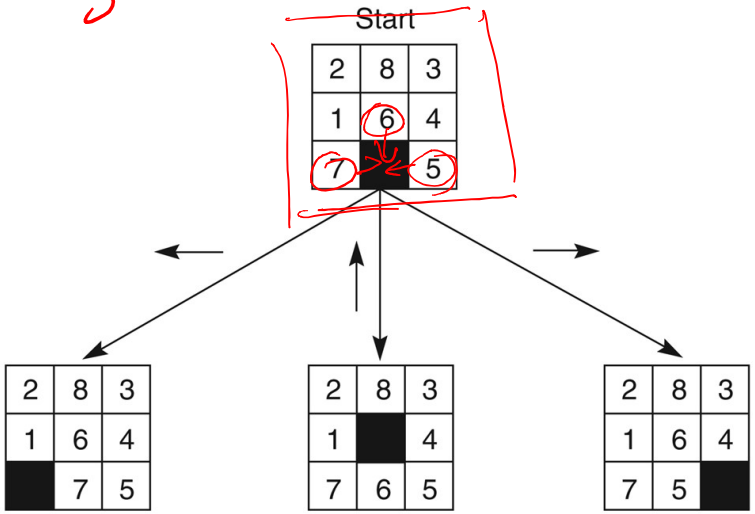
vs

$h_2(n)$ // much closer estimate to the actual cost but slow to run

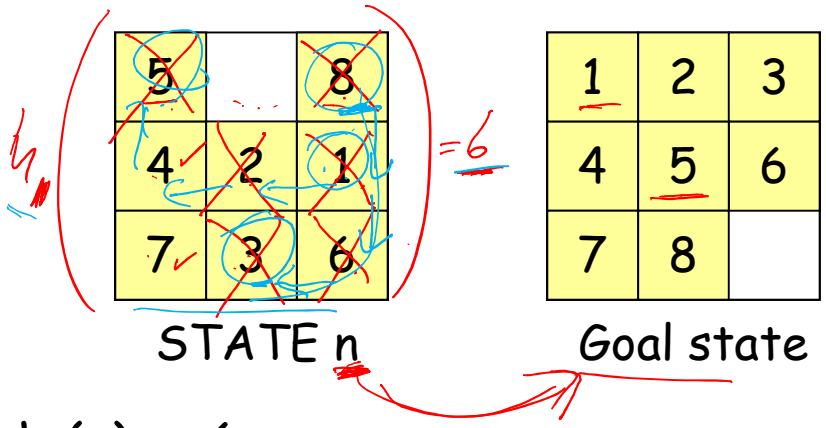
trade-off between length of the search path and total execution time

Example: 8-Puzzle – Heuristic 1

key: relax constraints of the problem to simplify it



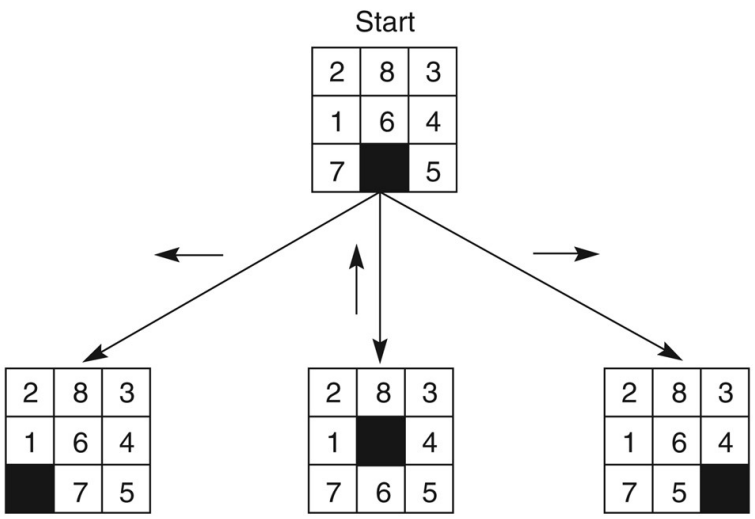
- h_1: Simplest heuristic (numbered)
  - Hamming distance : count number of tiles out of place when compared with goal



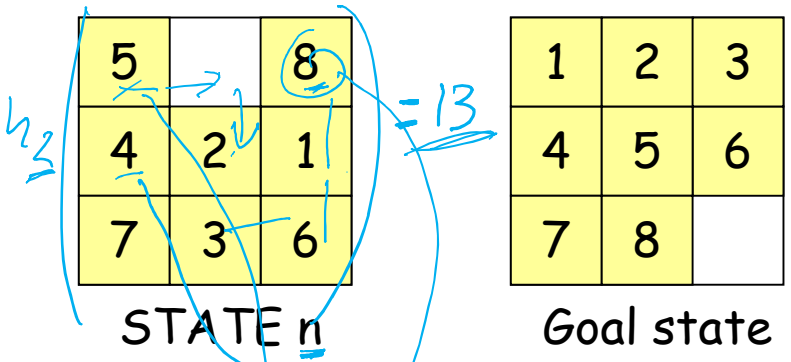
- $h_1(n) = 6$
  - does not consider the distance tiles have to be moved

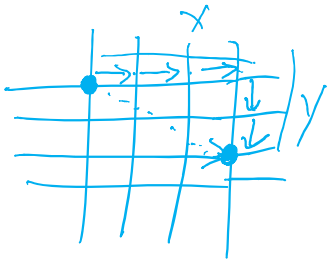
source: G. Luger (2005)

Example: 8-Puzzle – Heuristic 2



- h_2: Better heuristic
  - Manhattan distance: sum up all the distances by which tiles are out of place

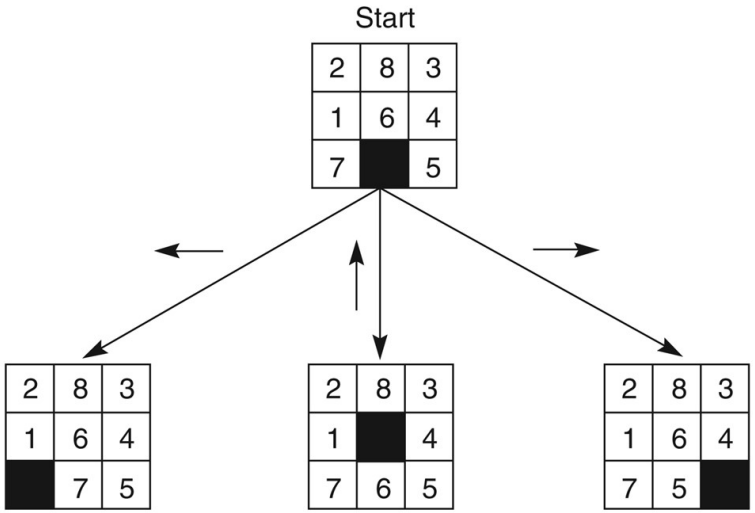




- $h_2(n) = 2+3+0+1+3+0+3+1$
  $= 13$

source: G. Luger (2005)

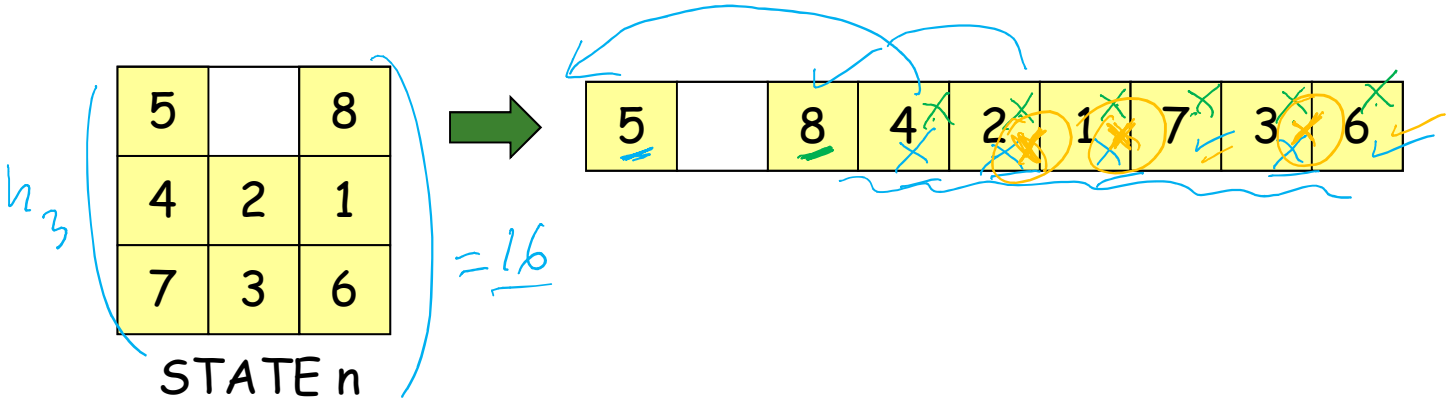
Example: 8-Puzzle – Heuristic 3



- h_3: Even Better
  - sum of permutation inversions
  - See next slide…

source: G. Luger (2005)

$h_3(N)$ = sum of permutation inversions



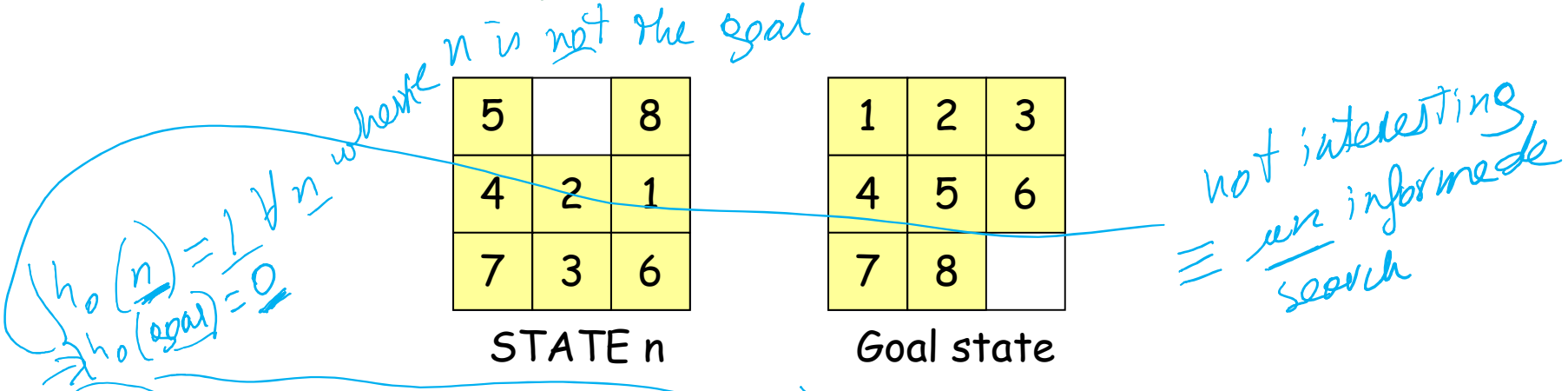
- For each numbered tile, count how many tiles on its right should be on its left in the goal state.

- $h_3(n) = n_5 + n_8 + n_4 + n_2 + n_1 + n_7 + n_3 + n_6$
  $= 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0$
  $= 16$

1	2	3
4	5	6
7	8	

Goal state

Heuristics for the 8-Puzzle



- $h_1(n)$ = misplaced numbered tiles
  = 6

- $h_2(n)$ = Manhattan distance
  = 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13

- $h_3(n)$ = sum of permutation inversions
  = $n_5 + n_8 + n_4 + n_2 + n_1 + n_7 + n_3 + n_6$
  = 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0 = 16

is $h_3(n)$ better?

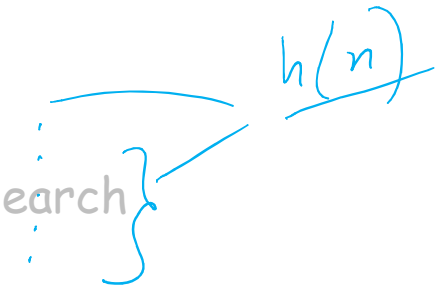
- $h_3(n)$ may return a better estimate ≈ closer to the actual cost
  - -> less backtracking / shorter search path
- BUT, $h_3(n)$ may be longer to compute // longer execution time
  - maybe overall longer to get the solution path
  - solution path may NOT be necessarily of lower cost (*more on this later*)

Today

1. State Space Representation ✓
2. State Space Search ✓
   a) Overview ✓
   b) Uninformed search ✓
      1. Breadth-first and Depth-first ✓
      2. Depth-limited Search ✓
      3. Iterative Deepening ✓
      4. Uniform Cost ✓
   c) **Informed search**
      1. **Intro to Heuristics** ✓
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

Up Next

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) **Informed search**
      1. **Intro to Heuristics**
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary



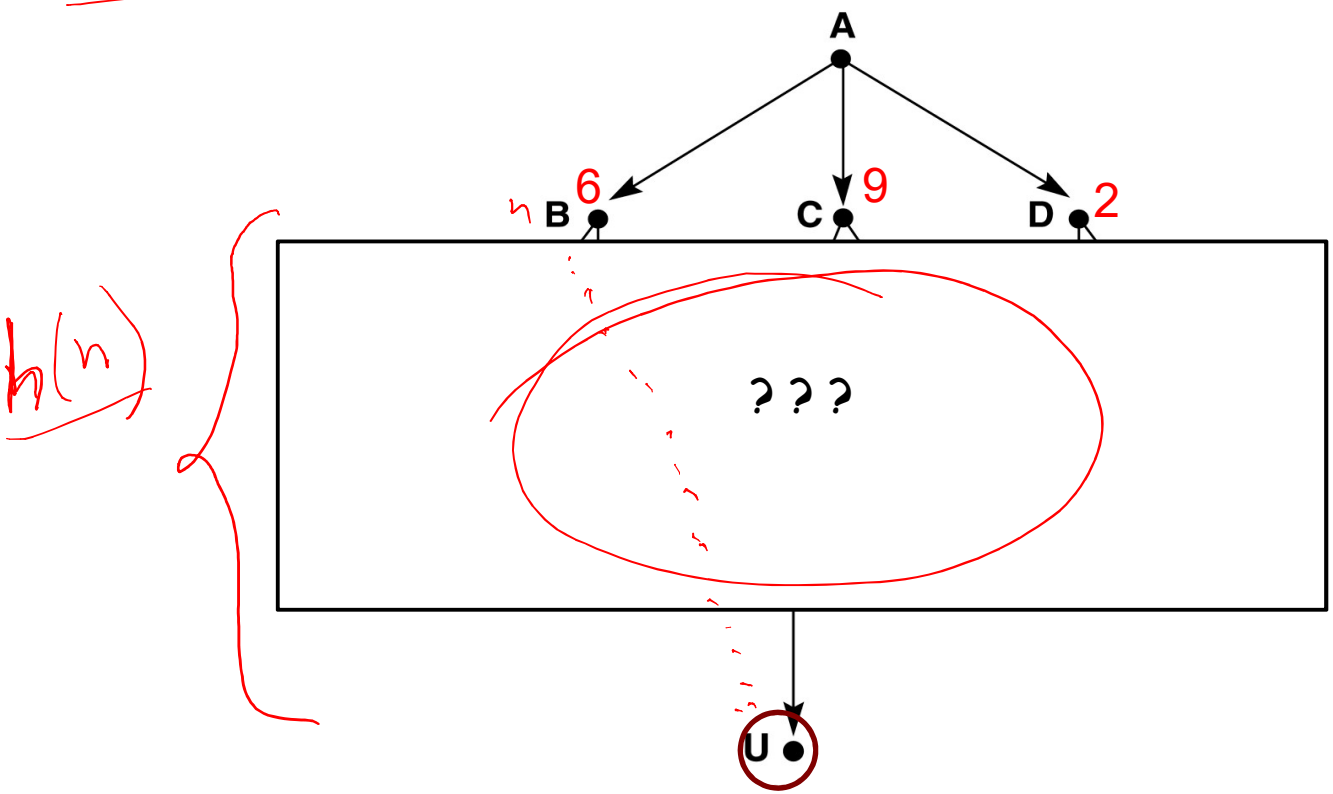
Artificial Intelligence: State Space Search part 3
Informed Search
Hill Climbing video #4

- Russell & Norvig – Sections 3.5.1, 3.5.2, 4.1.1

Today

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first Search and Depth-first Search
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill Climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

h(n)



- **h(n)** = estimate of the lowest cost from *n* to *goal*

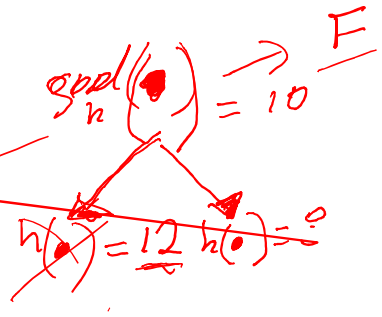
Hill Climbing

- General idea:
  - Similar to climbing a mountain in the fog with amnesia …
  - in the fog
    - --> only 1-step view of what is to come, so
    - if next step seems higher than where you are now -> go
    - otherwise, you assume you are at the top of the mountain -> stop
  - with amnesia -->
    - if you ever want to try other path, you can't because you did not keep track of where you came from

Vanilla HC vs Steepest Ascent HC

General Hill Climbing

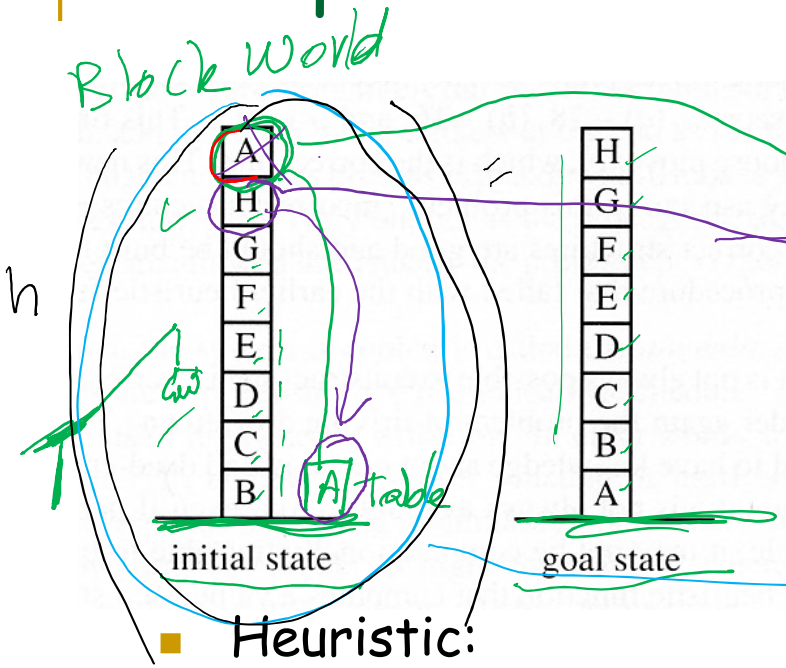
- uses h(n)
- does not use an open list (amnesia)



1. Vanilla Hill Climbing
   - take 1st successor s with better h() than current state n
   - i.e. if lower h(n) is better, chose 1st s with h(s) < h(n) // deep diving

2. Steepest ascent hill climbing:
   - generate all successor states S
   - run h() on all s ∈ S
   - among all successors s with better h() than current state n, take the successor s with the best h(n)

Example: Hill Climbing

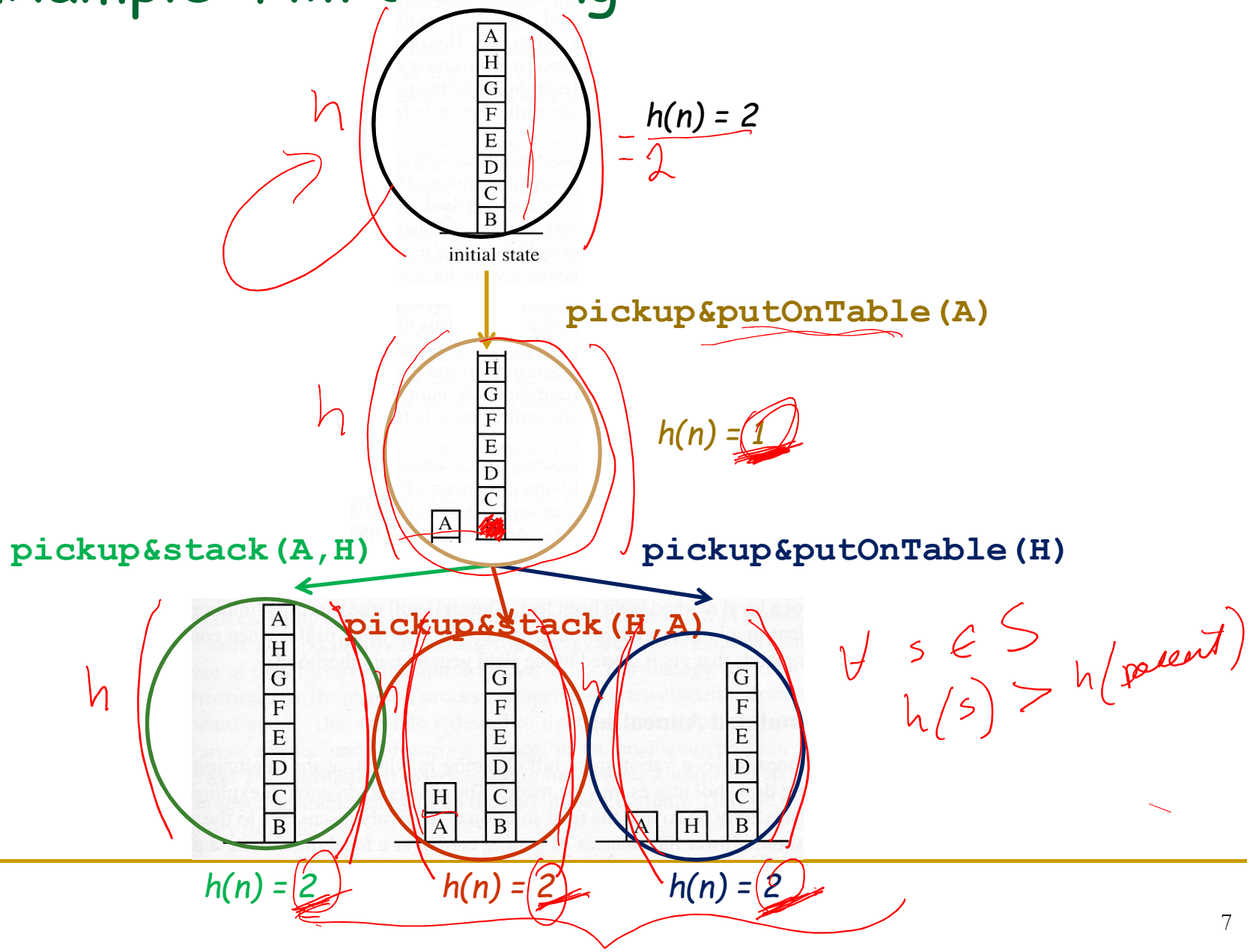


- Operators:
  - pickup&putOnTable(Block)
  - pickup&stack(Block1,Block2)

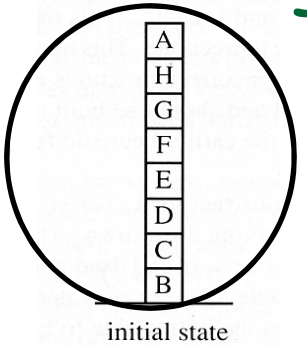
- Heuristic:
  - 0pt if a block is sitting where it is supposed to sit
  - +1pt if a block is NOT sitting where it is supposed to sit
  - so lower h(n) is better
    - h(initial) = 2
    - h(goal) = 0

source: Rich & Knight, Artificial Intelligence, McGraw-Hill College 1991.

Example: Hill Climbing

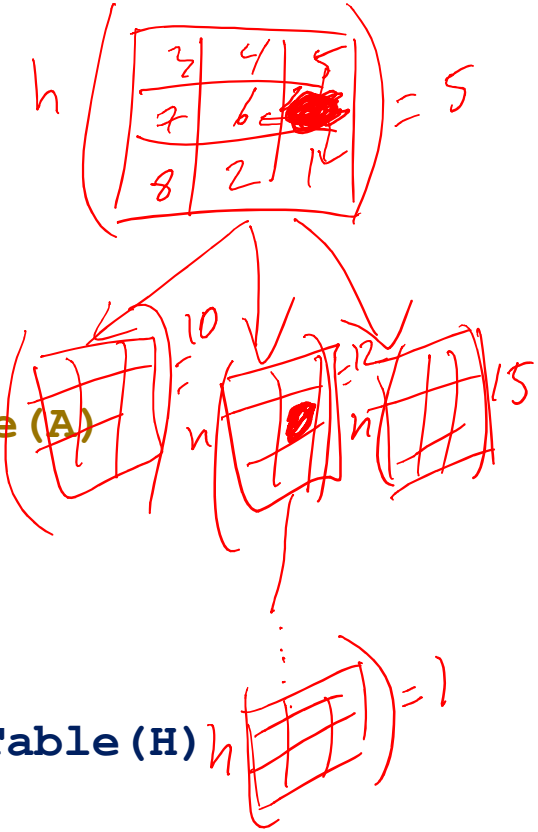


Example: Hill Climbing



h(n) = 2

pickup&putOnTable(A)



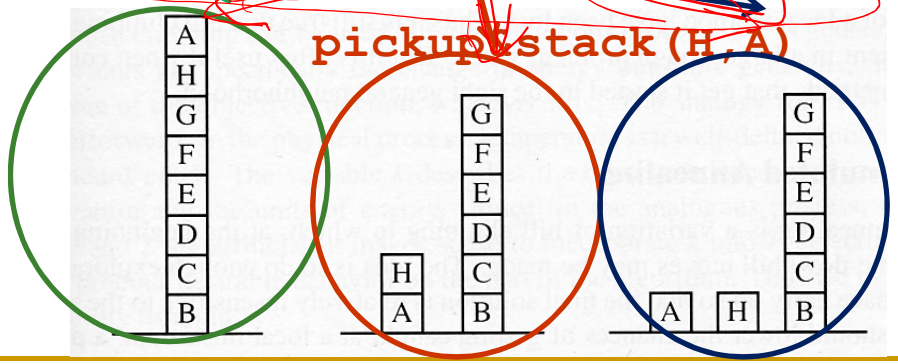
hill-climbing will stop, because all children have higher h(n) than the parent… --> local minimum

h(n) = 1

pickup&stack(A,H)

pickup&putOnTable(H)

pickup&stack(H,A)



h(n) = 2 h(n) = 2 h(n) = 2

Don't be confused… a lower h(n) is better…

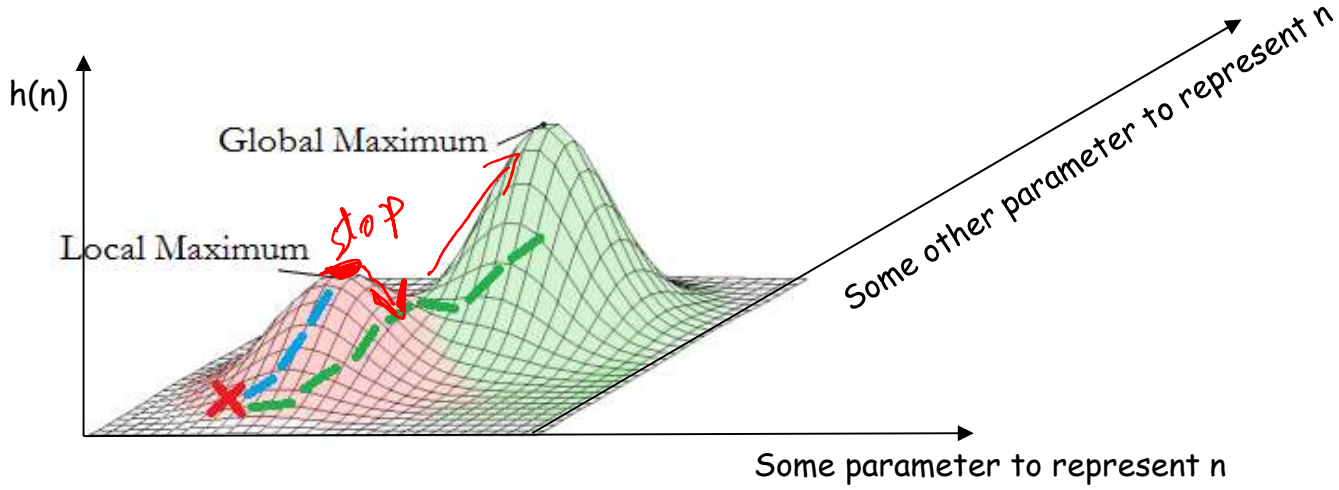
Steepest Ascent Hill Climbing

```
currentNode = startNode;
    loop do
        L = CHILDREN(currentNode);
        nextEval = +INFINITY;
        nextNode = NULL;

        for all c in L
           if (HEURISTIC-VALUE(c) < nextEval) // lower h is better
                nextNode = c;
                nextEval = HEURISTIC-VALUE(c);

          if nextEval >= HEURISTIC-VALUE(currentNode)
             // Return current node since no better child state exist
             return currentNode;

          currentNode = nextNode;
```

Adapted from https://en.wikipedia.org/wiki/Hill_climbing

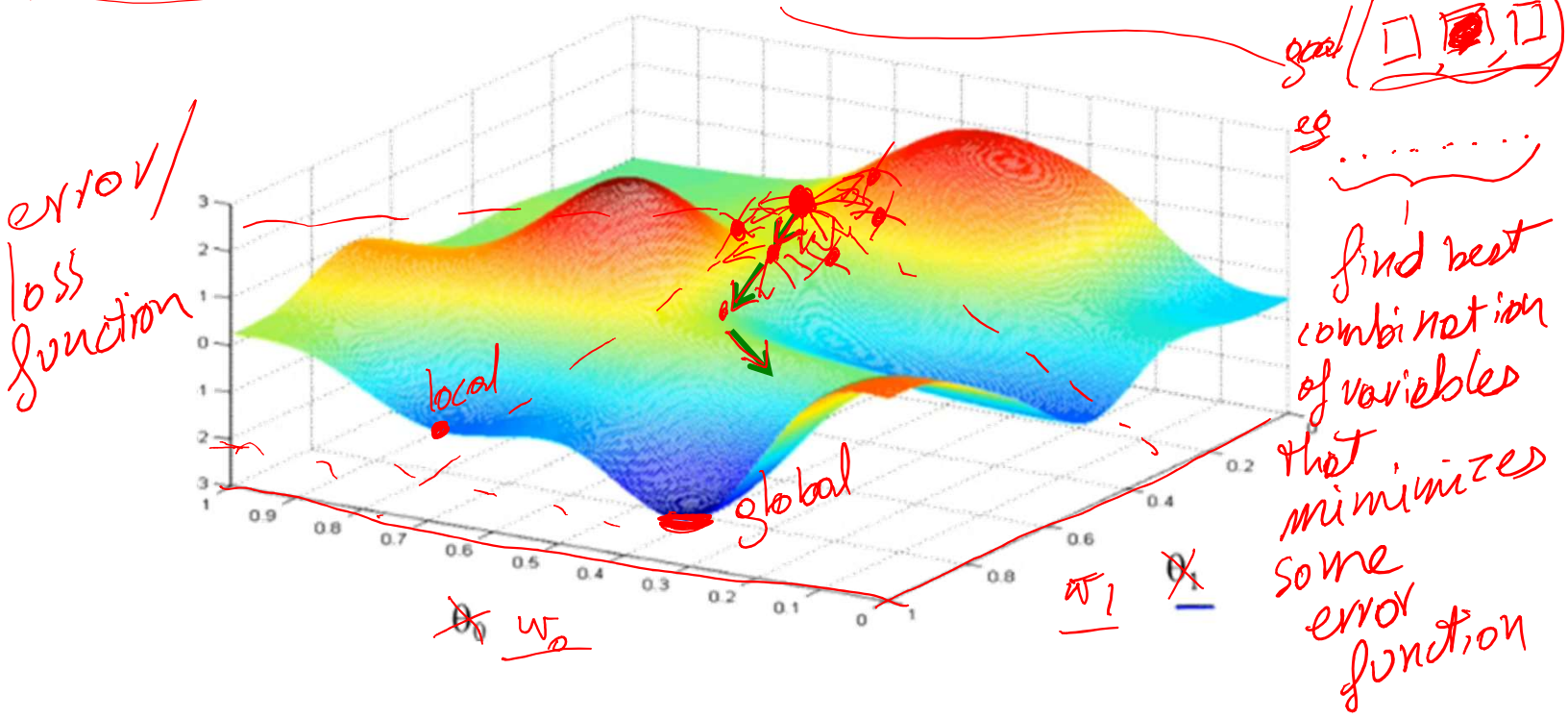
Problems with Hill Climbing

- Foothills (or local maxima)
  - reached a local maximum, not the global maximum
  - a state that is better than all its neighbors but is not better than some other states farther away.
  - at a local maximum, all moves appear to make things worse.
  - ex: 8-puzzle: we may need to move tiles temporarily out of goal position in order to place another tile in goal position



Use of Hill Climbing

- mostly for optimization problems
- i.e. goal defined not as a function of the state alone, but with respect of other states (eg. the best state I can reach)



Source: Andrew Ng

Today

1. State Space Representation ✓
2. State Space Search ✓
   a) Overview ✓
   b) Uninformed search ✓
      1. Breadth-first Search and Depth-first Search ✓
      2. Depth-limited Search ✓
      3. Iterative Deepening ✓
      4. Uniform Cost ✓
   c) Informed search ✓
      1. Intro to Heuristics ✓
      2. Hill climbing ✓
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

h(n)

Problem with Hill-Climbing

- used mostly for optimization problems
  - where the goal state is defined with respect to other states
  - ex. shortest path, longest....
- if goal state is independent of other states
  - we should be able to backtrack, and find another path to the goal
  - i.e. we should use an OPEN list
  - i.e. Greedy Best First Search

Up Next

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

Artificial Intelligence: State Space Search
Informed Search
Greedy Best First Search and Algorithms A and A*

part 3

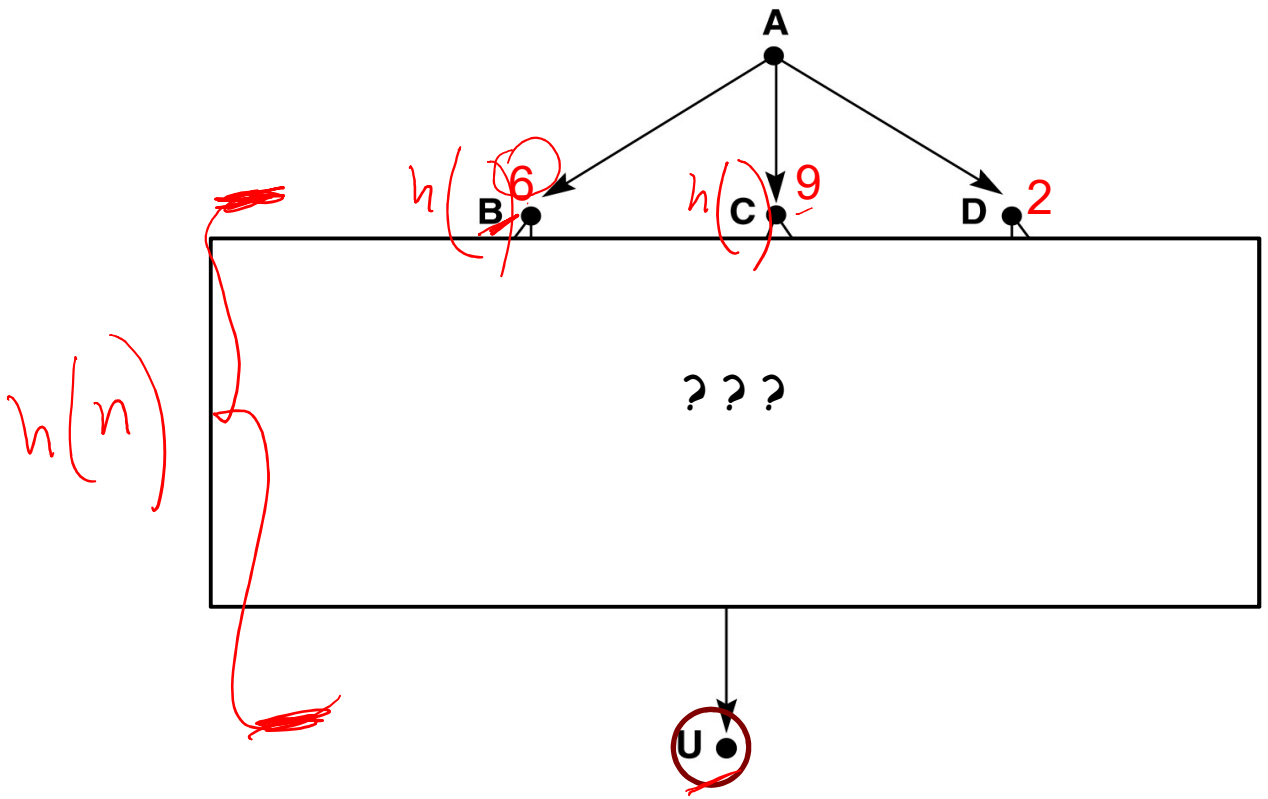
video #5

- Russell & Norvig – Sections 3.5.1, 3.5.2, 4.1.1

Today

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first Search and Depth-first Search
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

h(n)



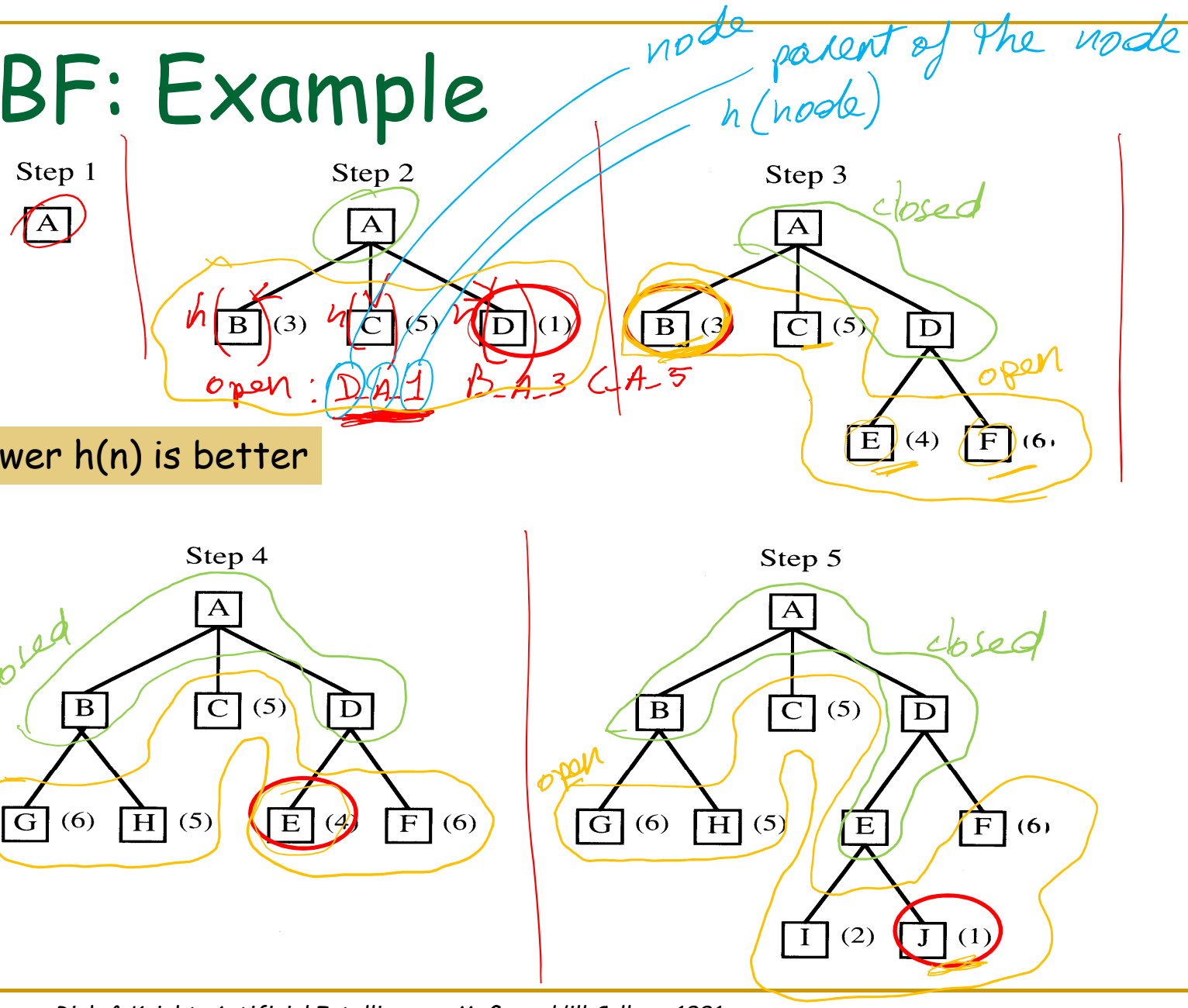
- **h(n)** = estimate of the lowest cost from *n* to *goal*

Greedy Best-First Search

- problem with hill-climbing:
  - no open list
  - --> can't backtrack
  - one move is selected and all others are forgotten
- solution to hill-climbing:
  - use "open" as a priority queue
  - this is called best-first search

- Best-first search:
  - Insert nodes in *open* list so that the nodes are sorted in ascending **h(n)**
  - Always choose the next node to visit to be the one with the best h(n) -- regardless of where it is in the search space

GBF: Example



Lower h(n) is better

source: Rich & Knight, Artificial Intelligence, McGraw-Hill College 1991.

Notes on GBF

- If you have a good $h(n)$, best-first can find a solution very quickly
- The solution may not be the optimal one (lowest cost) but there is a good chance of finding it quickly

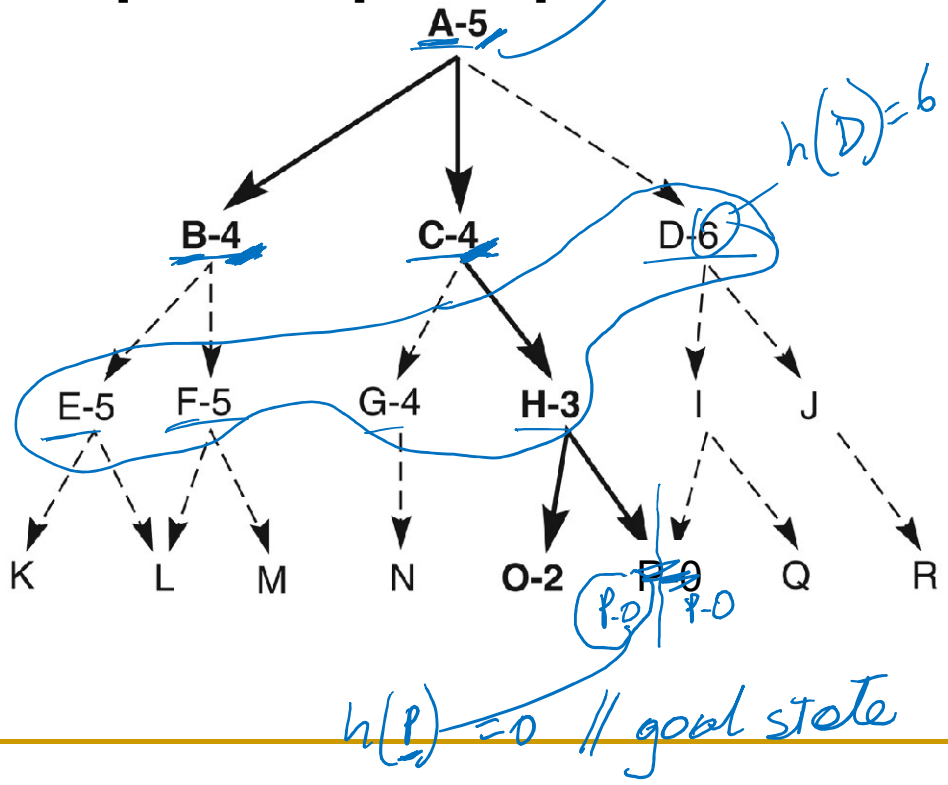
GBF Search: Example

1. open = [A-null-5] closed = []
2. open = [B-A-4 C-A-4 D-A-6] *(arbitrary choice)* closed [A]
3. open = [C-A-4 E-B-5 F-B-5 D-A-6] closed = [B A]
4. open = [H-C-3 G-C-4 E-B-5 F-B-5 D-A-6] closed = [C B A]
5. open = [P-H-0 0-H-2 G-C-4 E-B-5 F-B-5 D-A-6] closed = [H C B A]
6. goal P found

solution path: A C H P

priority queue sorted by h(n)

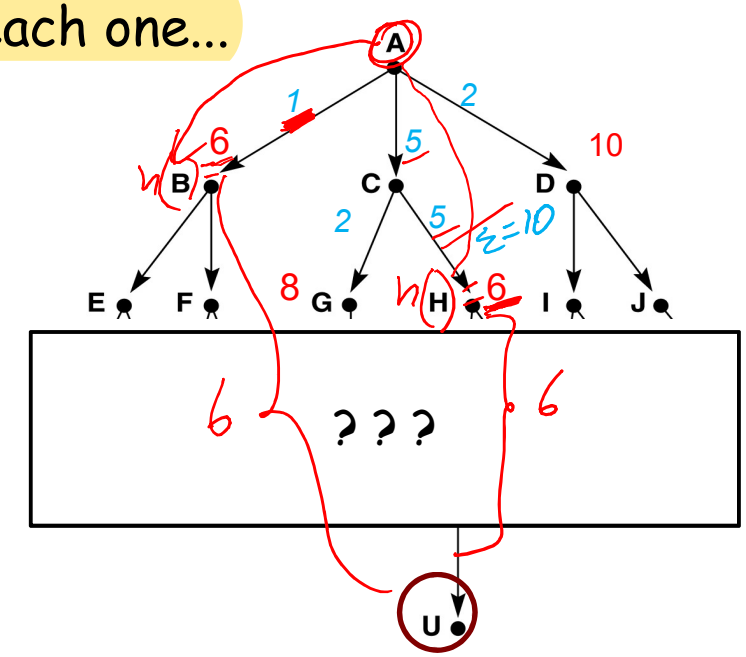
Lower h(n) is better

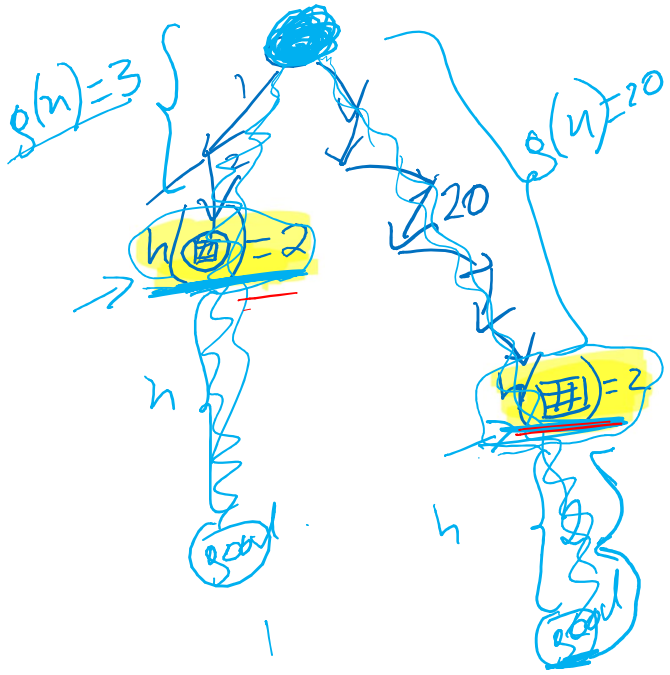


source: adapted from G. Luger (2005)

Problem with GBF search

- if 2 nodes have the same *h(n)*, no preference to the closest/least costly to reach one...





- Solution:
  - Maintain a cost count – *g(n)*
  - i.e. give preference to nodes with least expensive paths from root to n
  - i.e. combine *h(n)* and *g(n)*

Today

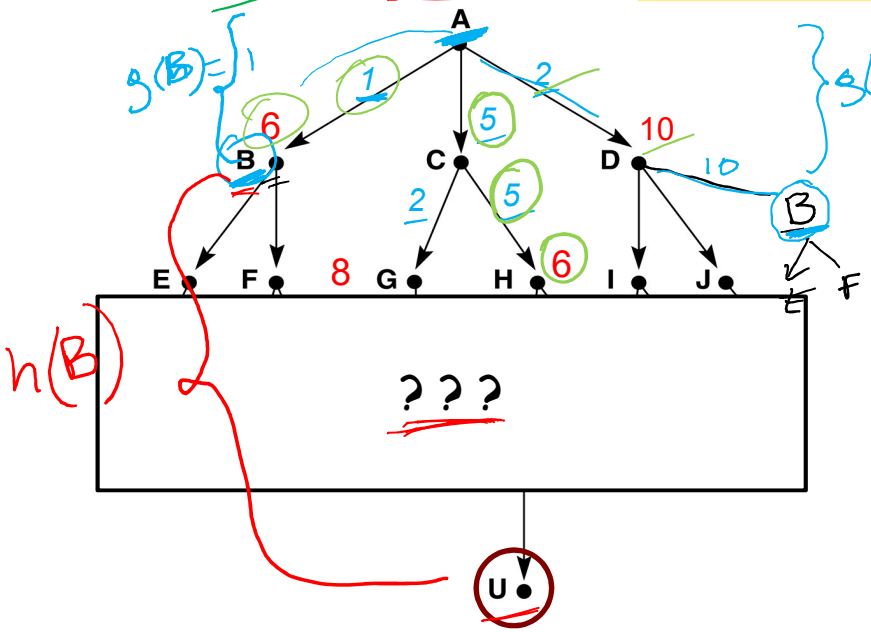
1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first Search and Depth-first Search
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

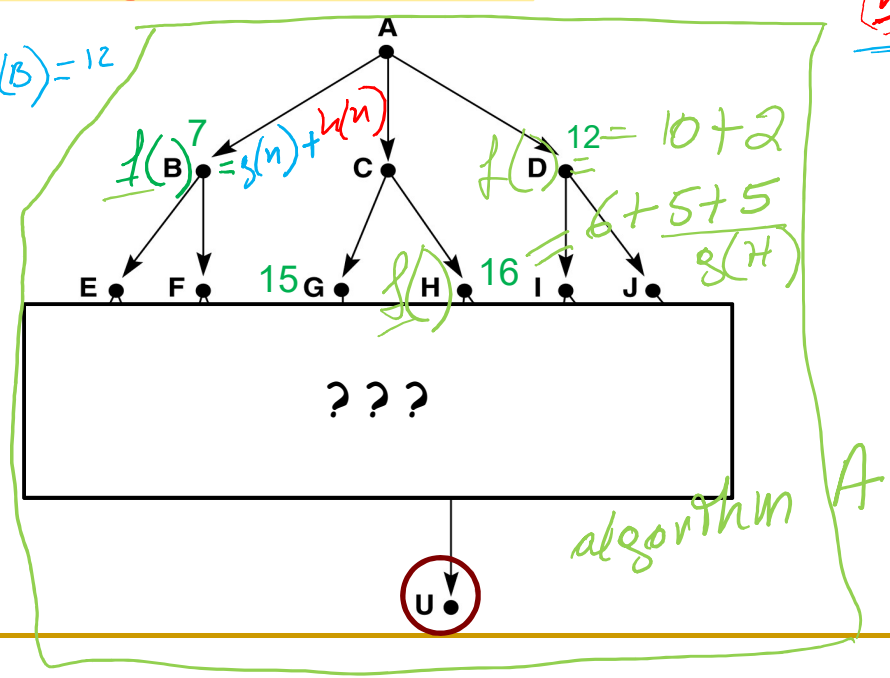
f(n) = h(n) + g(n)

- Modified evaluation function **f**:

  f(n) = g(n) + h(n)

  - f(n) estimate of total cost along path through n
  - g(n) actual cost of path from start to node n
  - h(n) estimate of cost to reach goal from node n





Algorithms A and A*

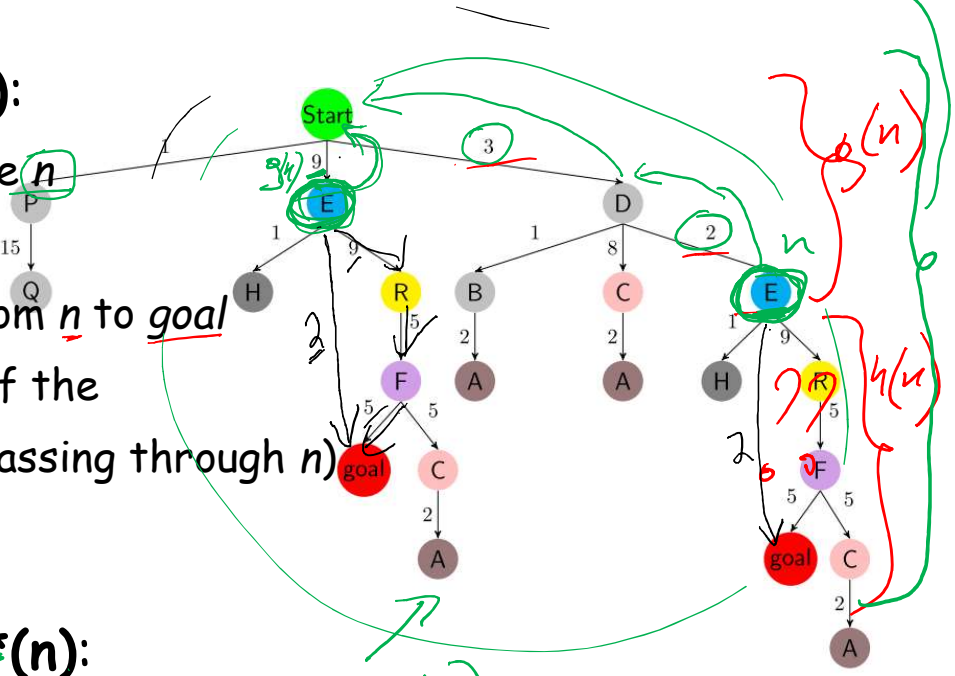
≠? (see next slides)

- similarly to Greedy Best first search:
  - keep an OPEN list as a priority queue
- But
  - OPEN is sorted by lowest f(n) = h(n) + g(n)

g(n)*, h(n)* and f*(n)

- We know that **f(n) = g(n) + h(n)**:
  - **g(n)** current cost from *start* to node *n*
    (*maybe not be the lowest cost*)
  - **h(n)** estimate of the lowest cost from *n* to *goal*

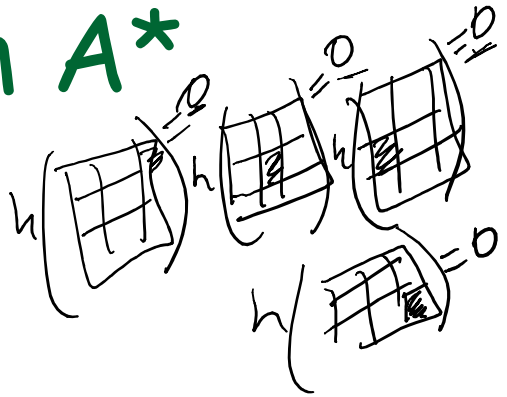
  --> **f(n)** estimate of the lowest cost of the solution path (from *start* to *goal* passing through *n*)



- Let us define **f*(n) = g*(n) + h*(n)**:
  - **g*(n)** cost of lowest cost path from *start* to node *n*
  - **h*(n)** actual lowest cost from *n* to *goal*

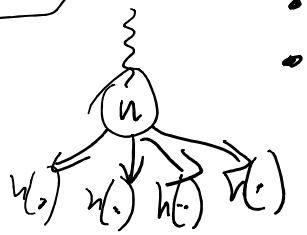
  --> **f*(n)** actual cost of lowest cost of the solution path (from *start* to *goal* passing through *n*)

Algorithm A vs Algorithm A*



- IF
  - $g(n) \geq g^*(n)$ for all n
    // ie. if the cost from the root to n is considered
- AND
  - $h(n) \leq h^*(n)$ for all n
    // ie. $h(n)$ never overestimates the true lowest cost from n to the goal
- THEN
  - algorithm A is called algorithm A*

uniform cost
- uses $g(n)$
- uses $h(n)$ where $h(n) = 0 \ \forall n$



what's the big deal?

--> algorithm A* is admissible

--> i.e. it guarantees to find the lowest cost solution path from the initial state to the goal

Algorithm A* vs GBF search

- given the same h(n):
  - *A* guarantees to find the lowest cost solution path*
  - *GBF does not*

- so is A* always "better" in real life?
  - not necessarily
  - computing g(n) can take time to compute
  - if client is not looking for the optimal (lowest cost) solution
  - a good-enough solution faster (i.e. GBF search) might be preferable

Today

1. State Space Representation ✓
2. State Space Search ✓
   a) Overview ✓
   b) Uninformed search ✓
      1. Breadth-first and Depth-first ✓
      2. Depth-limited Search ✓
      3. Iterative Deepening ✓
      4. Uniform Cost ✓
   c) Informed search
      1. Intro to Heuristics ✓
      2. Hill climbing ✓
      3. Greedy Best-First Search ✓
      4. Algorithms A & A* ✓
      5. More on Heuristics
   d) Summary

Up Next

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

next video

COMP 472 Artificial Intelligence
State Space Search
Informed Search part 3
More on Heuristics & Summary video 6

- Russell & Norvig – Section 3.5.2

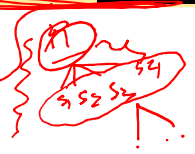
Today

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics

   d) Summary

Evaluating Heuristics

1. **Admissibility:**
   - "optimistic"
   - never overestimates the actual cost of reaching the goal
   - guarantees to find the lowest cost solution path to the goal (if it exists)



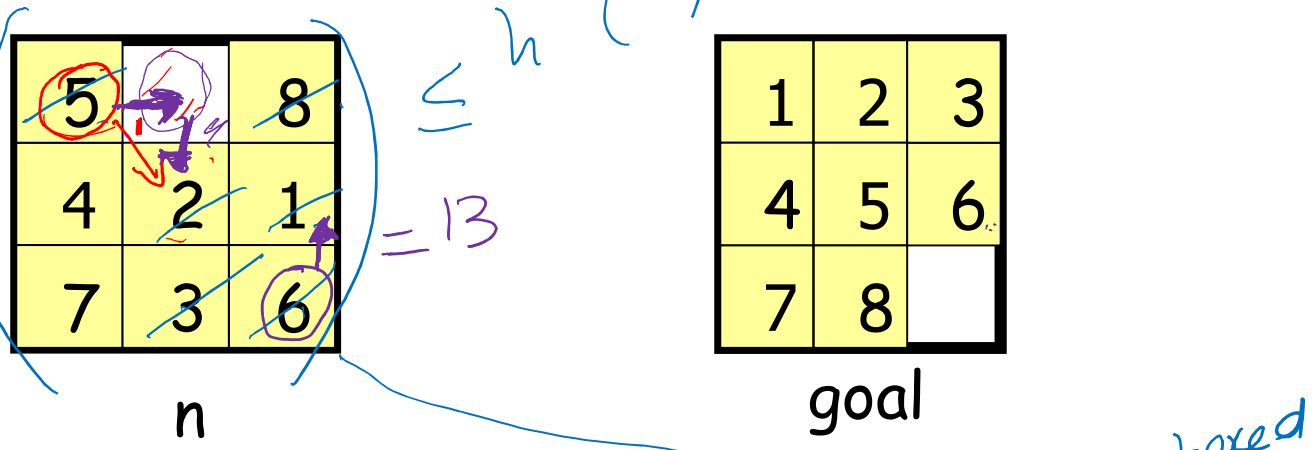
2. **Monotonicity:**
   - "local admissibility"
   - guarantees to find the lowest cost path to each state n visited (i.e. popped from OPEN)

3. **Informedness:**
   - measure for the "quality" of a heuristic
   - the more informed, the less backtracking, the shorter the search path

Admissibility

- A heuristic is admissible if it never overestimates the cost of reaching the goal
  - i.e.:
    - $h(n) \leq h^*(n)$ for all n
  - hence
    - $h(goal) = h^*(goal) = 0$
    - $h(n) = \infty$ if we cannot reach the goal from *n*
- Algorithm A that uses an admissible heuristic
  - is called algorithm A*
  - guarantees to find the lowest cost solution path to the goal (if it exists)
  - note: does not guarantee to find the lowest cost search path
  - e.g.: uniform cost is admissible -- it uses $f(n) = g(n) + 0$

Example: 8-Puzzle

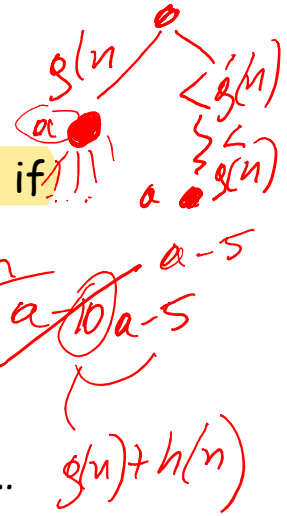


- h1(n) = Hamming distance = number of misplaced tiles = 6
--> admissible

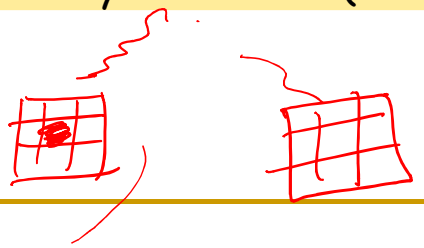
- h2(n) = Manhattan distance = 13
--> admissible

Problem with Admissibility

- Admissible heuristics may temporarily reach non-goal states along a suboptimal path
  - remember with uniform-cost… when we expanded a node, we had to check if it was already in the OPEN list with a higher path path, and if so, we would replace it with the current path cost/parent info



- With A*, if we have a node *n* in OPEN or even in CLOSED
  - We may later find *n* again, but with a lower *f(n)* (due to a lower *g(n)*… the *h(n)* will, by definition be the same).
  - So to ensure that the solution path has the lowest cost,
    - We may need to update the cost/parent info of node *n* in OPEN or even put *n* back in OPEN even if it has already be visited (i.e. in CLOSED)… expensive work…



Admissibility and A* Search

- Admissibility: $\forall n \; h(n) \leq h^*(n)$
  - To guarantee to find the lowest cost solution path, when we generate a successors s: i.e. when node n is popped from OPEN

1. IF s is already in CLOSED
   IF s in CLOSED has a higher f-value due to a higher g-value i.e g(s)
   THEN place s and its new lower f-value in OPEN !
   // we found a lower cost path to s, but we had already expanded s…
   // to guarantee the lowest cost solution path, we need to put s back in OPEN and re-visit it again
   ELSE ignore s

2. ELSE IF s is already in OPEN
   IF s in OPEN has a higher f-value
   THEN replace the old s in OPEN with the new lower f-value s
   // we found a lower cost path to s, and we had not expanded s yet
   // to guarantee the lowest cost solution path, we need to replace the old s in OPEN with the new
   // lower-cost s
   ELSE ignore s

3. ELSE insert s in OPEN
   // as usual

expensive but necessary if you want to guarantee lowest cost solution

Monotonicity (aka consistent)

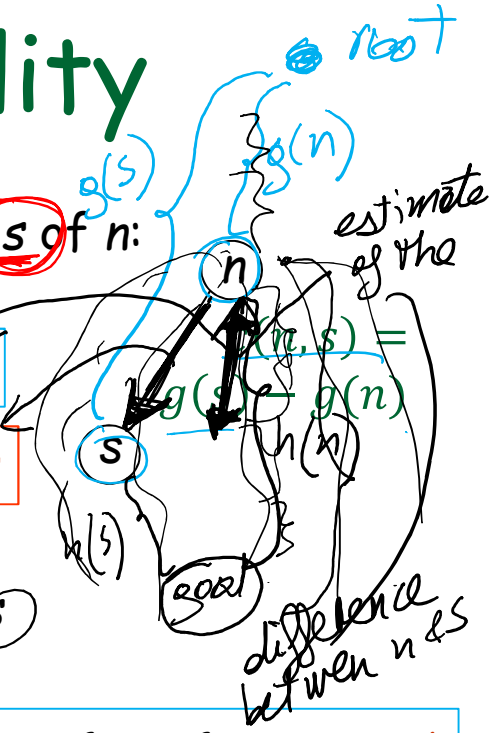
- Admissibility:
  - does not guarantee that every node *n* that is expanded (i.e. for which we generate the successors *s*) will have been found via the lowest cost.
- Monotonicity
  - guarantees that!
  - Stronger property than admissibility
- If a heuristic is monotonic
  - We are guaranteed that once a node is popped from the OPEN list, we have found the lowest cost path to it
  - i.e we always find the lowest cost path to each node, the 1st time it is popped from OPEN!
  - So once a node is placed in the CLOSED list, if we encounter it again, we do not need to check that the 2nd encounter has a lower cost. We can just ignore it. (more efficient!)

Monotonicity vs Admissibility

- *h* is monotonic if for every node *n* and every successor *s* of *n*:
  - $h(n) \leq c(n,s) + h(s) \quad \forall n, s$
  - $h(n) - h(s) \leq c(n,s)$
  - $h(n) - h(s) \leq g(s) - g(n)$

Estimate of cost from n to s

Actual of cost from n to s



- -> monotonic = *h(n)* is optimistic for all transitions *n*-->*s*
- *f(n)* is non-decreasing along any path

- admissibility = *h(n)* only needs to be optimistic for *n*-->*goal*
  - $h(n) \leq h^*(n) \quad \forall n$
  - $h^*(n) = g(goal) - g(n) \rightarrow h(n) \leq g(goal) - g(n)$
  - $h(goal) = 0 \rightarrow h(n) - h(goal) \leq g(goal) - g(n)$

Estimate of cost from n to goal

Actual of cost from n to goal

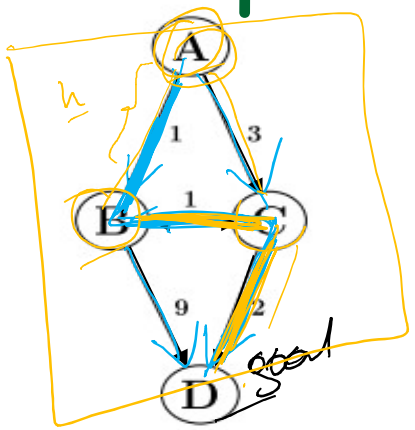
- Every monotonic *h(n)* is admissible (but not vice-versa)

Monotonicity and A* Search

- Monotonicity
  - Guarantees to find the lowest cost solution path
  - Guarantees to find the lowest cost path to every node, the first time we expand it.
  - --> no need to check the CLOSED list again!
  - So when we generate a successors s:

1. ~~IF s is already in CLOSED~~
   ~~IF s in CLOSED has a higher f-value~~
   ~~THEN place s and its new lower f-value in OPEN !~~
   ~~// we found a lower cost path to s, but we had already expanded s…~~
   ~~// to guarantee the lowest cost solution path, we need to put s back in OPEN and re-visit it again~~
   ~~ELSE ignore s~~
2. ELSE IF s is already in OPEN
   IF s in OPEN has a higher f-value
   THEN replace the old s in OPEN with the new lower f-value s
   // we found a lower cost path to s, and we had not expanded s yet
   // to guarantee the lowest cost solution path, we need to replace the old s in OPEN with the new lower-
   // cost s
   ELSE ignore s
3. ELSE insert s in OPEN
   // as usual

Example



node	h_1	h_2
A	4	4
B	3	3
C	2	0
D	0	0

Solution paths
1. A B D -> cost of 10
2. A C D -> cost of 5
3. A B C D -> cost of 4
4. A C B D -> cost of 13

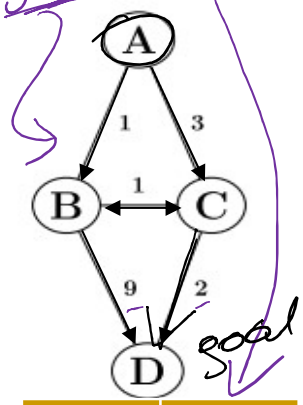
- Admissibility -- h*(A)=4 h*(B)=3 h*(C)=2 h*(D)=0
  - is h_1 admissible? Yes
  - is h_2 admissible? Yes

- Monotonic
  - is h_1 monotonic? Yes
    - $h_1(A) - h_1(B) \leq g(B) - g(A)$ $\quad 4 - 3 \leq 1 - 0$ $\quad 1 \leq 1$
    - $h_1(A) - h_1(C) \leq g(C) - g(A)$ $\quad 4 - 2 \leq 2 - 0$ $\quad 2 \leq 2$
    - $h_1(A) - h_1(D) \leq g(D) - g(A)$ $\quad 4 - 0 \leq 4 - 0$ $\quad 4 \leq 4$
    - ...
  - is h_2 monotonic? No
    - $h_2(A) - h_2(C) \nleq g(C) - g(A)$ $\quad 4 - 0 \nleq 2 - 0$ $\quad 3 \nleq 2$
    - $h_2(B) - h_2(C) \nleq g(C) - g(B)$ $\quad 3 - 0 \nleq 2 - 1$ $\quad 3 \nleq 1$

Example - h_2

node n
parent of n
$f(n) = g(n) + h(n)$
admissible + not monotonic

	OPEN (unsorted... work in progress)	OPEN	CLOSED
1	• $A_{null}4_{g=0+h=4}$	1. $A_{null}4_{g=0+h=4}$	
2	• $B_A4_{g=1+h=3}$ • $C_A3_{g=3+h=0}$ <sort>	1. $C_A3_{g=3+h=0}$ // we will explore C directly ~~from A, but~~ there is a lower cost path to C (ABC). h1 found it because it is monotonic, but h2 is not monotonic, so it could not guarantee that when we expand a node, we have found the lowest cost path to it... 2. $B_A4_{g=1+h=3}$	$A_{null}4_{g=0+h=4}$
3	• $B_C7_{g=3+1+h=3}$ // B is already in OPEN (see below) but with a lower cost path. We do not replace the old, and ignore this version • $D_C5_{g=3+2+h=0}$ • $B_A4_{g=1+h=3}$	1. $B_A4_{g=1+h=3}$ 2. $D_C5_{g=3+2+h=0}$	$A_{null}4_{g=0+h=4}$ $C_A3_{g=3+h=0}$
4	• $C_B2_{g=1+1+h=0}$ // C was already in CLOSED but with a higher f-value, we just found a lower cost path to C... we need to put this version back in OPEN ;-(• $D_B10_{g=1+9+h=0}$ // D is already in OPEN (see below) but with a lower cost path. We do not replace the old, and ignore this version • $D_C5_{g=2+2+h=0}$	1. $C_B2_{g=1+1+h=0}$ 2. $D_C5_{g=3+2+h=0}$	$A_{null}4_{g=0+h=4}$ ~~$C_A3_{g=3+h=0}$~~ $B_A4_{g=1+h=3}$
5	• $B_C6_{g=1+1+1+h=3}$ • $D_C4_{g=1+1+2+h=0}$ // D is already in OPEN (see below) but with a higher cost path. We replace the old version with version • $D_C5_{g=1+9+h=0}$	1. $D_C4_{g=1+1+2+h=0}$ 2. $B_C6_{g=1+1+1+h=3}$	$A_{null}4_{g=0+h=4}$ $B_A4_{g=1+h=3}$ $C_B2_{g=1+1+h=0}$
		goal(D_C4) = true! Solution path = $D_C C_B B_A A_{null}$ cost = 2+1+1 = 4 // lowest cost path found in 5 steps	

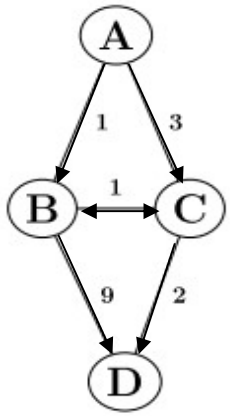


node	h_2
A	4
B	3
C	0
D	0

lowest cost path found in 5 steps

Example – h_1 admissible + monotonic

	OPEN (unsorted… work in progress)	OPEN	CLOSED
1	• $A_{null}4_{g=0+h=4}$	1. $A_{null}4_{g=0+h=4}$	
2	• $B_A4_{g=1+h=3}$ • $C_A5_{g=3+h=2}$	1. $B_A4_{g=1+h=3}$ 2. $C_A5_{g=3+h=2}$	$A_{null}4_{g=0+h=4}$
3	• $C_B4_{g=1+1+h=2}$ *// C already in OPEN with a higher f-value, replace old version with this one* • $D_B10_{g=1+9+h=0}$ • $C_A5_{g=3+h=2}$	1. $C_B4_{g=1+1+h=2}$ 2. $D_B10_{g=1+9+h=0}$	$A_{null}4_{g=0+h=4}$ $B_A4_{g=1+h=3}$
4	• $D_C4_{g=2+2+h=0}$ *// D already in OPEN with a higher f-value, replace old version with this one* • $B_C6_{g=2+1+h=3}$ *// B already in CLOSED but since h1 is monotonic, we do not need to check the f-value of the version in CLOSED because we know that the version in CLOSED will have a lower f-value, so can ignore this version* • $D_B10_{g=1+9+h=0}$	1. $D_C4_{g=2+2+h=0}$ 2. $B_C6_{g=2+1+h=3}$	$A_{null}4_{g=0+h=4}$ $B_A4_{g=1+h=3}$ $C_B4_{g=1+1+h=2}$
		goal(D_C4) = true! Solution path = $D_C\,C_B\,B_A\,A_{null}$ cost = 2+1+1 = 4 *// lowest cost path found in 4 steps*	

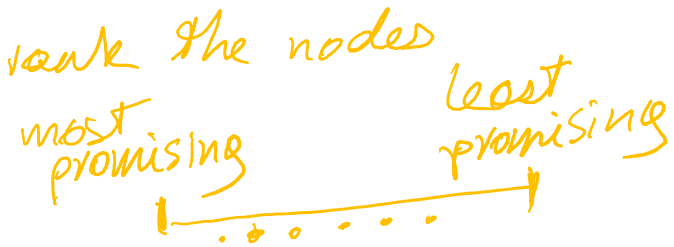


node	h_1
A	4
B	3
C	2
D	0

Admissible + Monotonic

lowest cost path found in 4 steps

Informedness



- Intuition:
  - h(n) = 0 for all nodes is less informed
  - number of misplaced tiles is less informed than Manhattan distance
- Formally:
  - given 2 admissible heuristics h_1 and h_2 *// ie. $h_1(n) \le h^*(n)$ and $h_2(n) \le h^*(n)$* $\forall n$
    - if $h_1(n) \le h_2(n)$, for all states n
    - then h_2 is more informed than h_1
    - aka h_2 dominates h_1
- So?
  - a more informed heuristic expands fewer nodes
  - aka the search path is shorter
  - however, you need to consider the computational cost of evaluating the heuristic… $h(n)$
  - the time spent computing heuristics must be recovered by a better search

Today

1. State Space Representation
2. State Space Search
   a) Overview
   b) Uninformed search
      1. Breadth-first and Depth-first
      2. Depth-limited Search
      3. Iterative Deepening
      4. Uniform Cost
   c) Informed search
      1. Intro to Heuristics
      2. Hill climbing
      3. Greedy Best-First Search
      4. Algorithms A & A*
      5. More on Heuristics
   d) Summary

Summary

Search	Uses h(n)?	Uses g(n)?	OPEN list
Breadth-first	No	No	Priority queue sorted by level
Depth-first	No	No	Stack
Depth-limited	No	No	Stack
Iterative Deepening	No	No	Stack
Uniform Cost *- guarantees to find the lowest cost solution path*	No	Yes	Priority queue sorted by g(n) When generating successors: - If successor s already in OPEN with higher g(n), replace old version with new s - If successor s already in CLOSED, ignore s
Hill Climbing	Yes	No	N/A
Greedy Best-First *- no constraints on h(n)* *- no guarantee to find lowest cost solution path*	Yes	No	Priority queue sorted by h(n)
Algorithm A *- no constraints on h(n)* *- no guarantee to find lowest cost solution path*	Yes	Yes	Priority queue sorted by f(n)
Algorithm A* *- h(n) must be admissible* *- guarantees to find the lowest cost solution path*	Yes	Yes	Priority queue sorted by f(n) If h(n) is NOT monotonic When generating successors: - If successor s already in OPEN with higher f(n), replace old version with new s - If successor s already in CLOSED with higher f(n), replace old version with new s If h(n) IS monotonic When generating successors: - If successor s already in OPEN with higher f(n), replace old version with new s - If successor s already in CLOSED, ignore it.

Today

1. State Space Representation ✓
2. State Space Search ✓
   a) Overview ✓
   b) Uninformed search ✓
      1. Breadth-first and Depth-first ✓
      2. Depth-limited Search ✓
      3. Iterative Deepening ✓
      4. Uniform Cost ✓
   c) Informed search ✓
      1. Intro to Heuristics ✓
      2. Hill climbing ✓
      3. Greedy Best-First Search ✓
      4. Algorithms A & A* ✓
      5. More on Heuristics ✓
   d) Summary ✓

Up Next

1. Part 4: Adversarial Search

Artificial Intelligence: Adversarial Search part 4 Minimax

- Russell & Norvig: Chapter 5

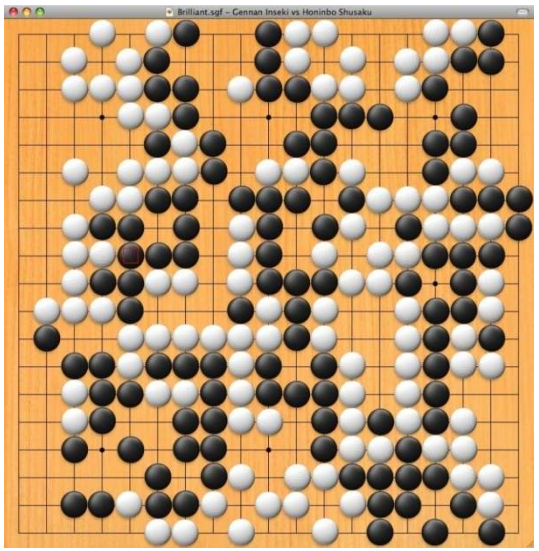
Today

- Adversarial Search 

1. Minimax 4.1
2. Alpha-beta pruning 4.2
3. Other Adversarial Search 4.3
   1. Multiplayer Games
   2. Stochastic Games
   3. Monte Carlo Tree Search

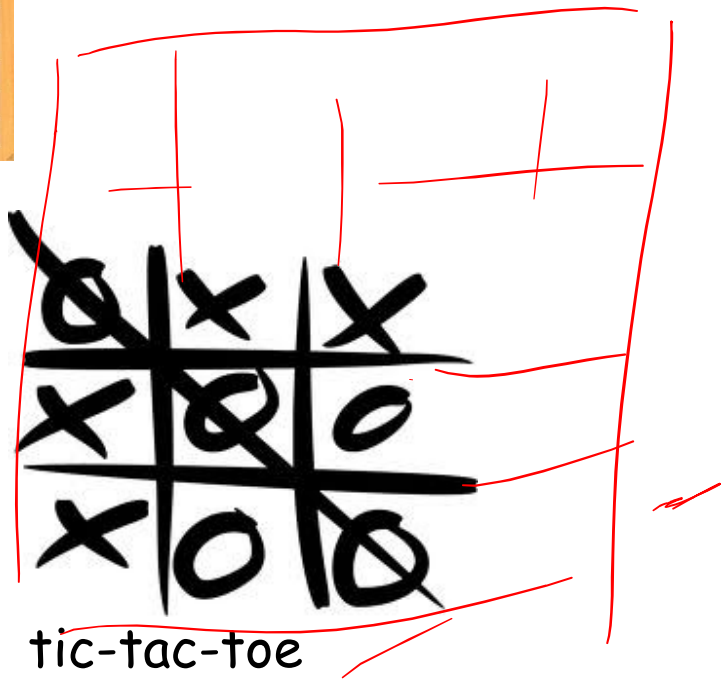
Chap 5

Motivation



GO

chess



tic-tac-toe

Games

1. Assumptions
   1. Zero-sum game:
      - no player has a game advantage. If the total gains of one player are added up, and the total losses are subtracted, they will sum to 0.
   2. Players play rationally (i.e. to win)

2. Characteristics of Games
   - Number of players? ~~1~~, 2, 3+
   - Deterministic:
     - the outcome of the game is only dependent on the moves of the players
   - Stochastic:
     - chance involved
   - Perfect information:
     - all players know the state of the game and all possible moves a.k.a *fully observable*
   - Imperfect information:
     - eg. a player is hiding their game aka. *partially observable*

	Deterministic	Stochastic
Perfect information	chess, checkers, go	backgammon, monopoly
Imperfect information	battleship	scrabble, poker, bridge

Techniques

Nb of players	Deterministic / Stochastic ?	Perfect / Imperfect information?	Example	Technique
1	deterministic	perfect info	8-puzzle	heuristic search
2	deterministic	perfect info	chess, checkers, go	minimax & alpha-beta
3+	deterministic	perfect info	Chinese Checkers	max^n
1	stochastic	yes	2048	expectimax
2	stochastic	yes	backgammon	expectiminimax
3+	stochastic	yes		modified expectimax

Techniques

Nb of players	Deterministic / Stochastic ?	Perfect / Imperfect information?	Example	Technique
1	deterministic	perfect info	8-puzzle	heuristic search
2	deterministic	perfect info	chess, checkers, go	minimax & alpha-beta
3+	deterministic	perfect info	Chinese Checkers	max^n
1	stochastic	yes	2048	expectimax
2	stochastic	yes	backgammon	expectiminimax
3+	stochastic	yes		modified expectimax

Today

- Adversarial Search
  1. Minimax 

  2. Alpha-beta pruning
  3. Other Adversarial Search
     1. Multiplayer Games
     2. Stochastic Games
     3. Monte Carlo Tree Search

Minimax Search

- Game between two opponents, MIN and MAX
  - MAX tries to win, and
  - MIN tries to minimize MAX's score
- Existing heuristic search methods do not work
  - would require a helpful opponent
  - need to incorporate "hostile" moves into search strategy

- 2 flavors:

1. exhaustive Minimax

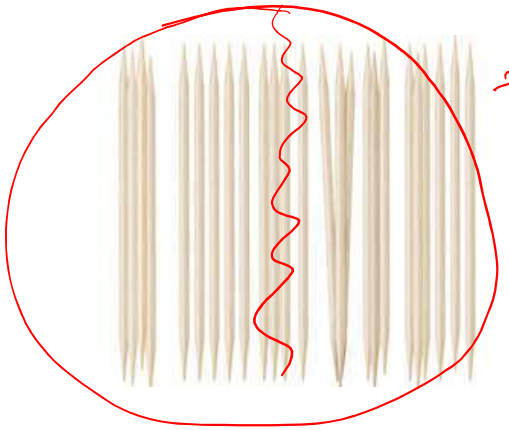
2. n-ply Minimax with Heuristic

Exhaustive Minimax Search

- For small games where exhaustive search is feasible
- Procedure:
  1. build complete game tree
  2. label each level according to player's turn (MAX or MIN)
  3. label leaves with a utility function to determine the outcome of the game
     - e.g., (0, 1) or (-1, 0, 1)
  4. propagate this value up:
     - if parent=MAX, give it max value of children
     - if parent=MIN, give it min value of children
  5. Select best next move for player at the root as the move leading to the child with the highest value (for MAX) or lowest values (for MIN)

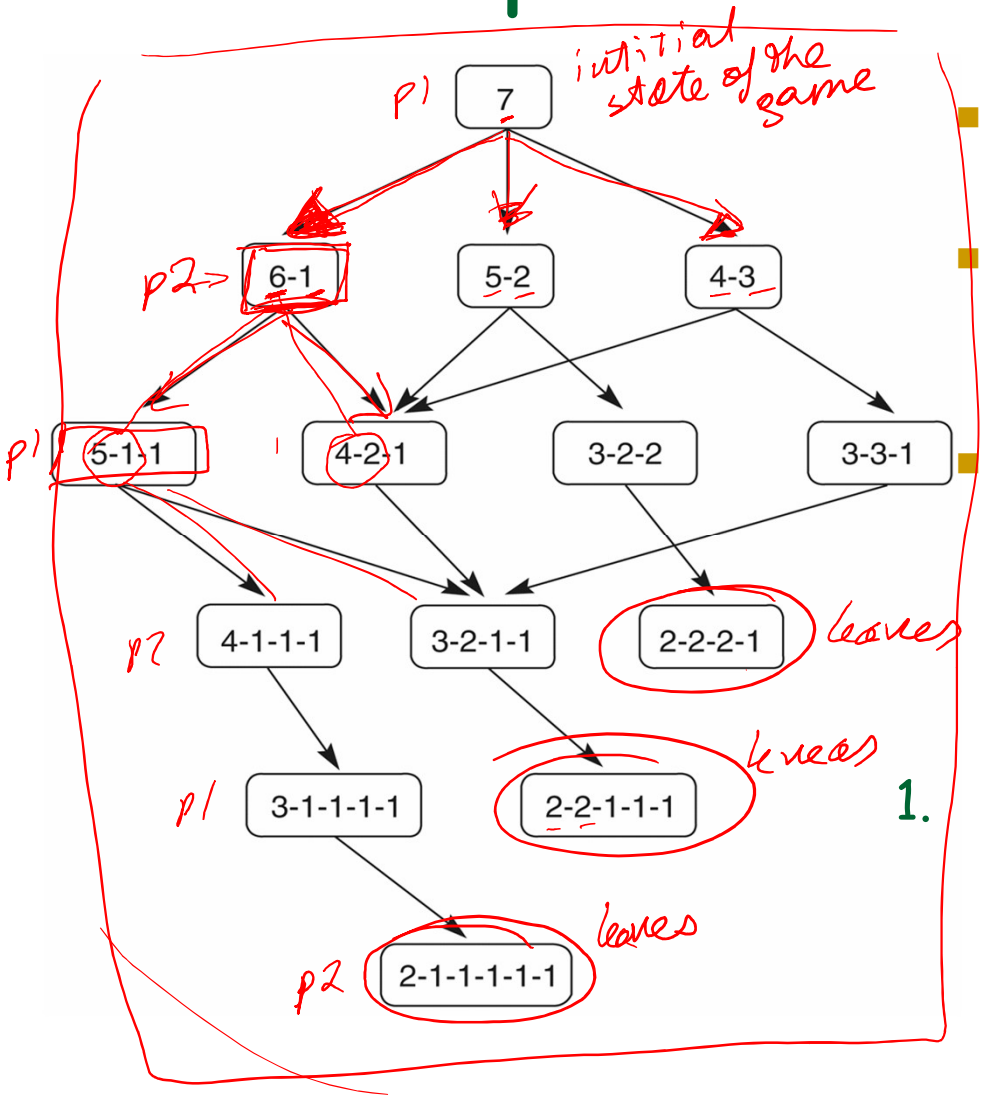
Example: Game of Nim

- Rules
  - 2 players start with a pile of tokens
  - move: split (any) existing pile into two non-empty differently-sized piles
  - game ends when no pile can be unevenly split
  - player who cannot make their move loses



https://www.scholastic.com/parents/kids-activities-and-printables/printables/math-worksheets/game-nim.html

State Space of Game Nim

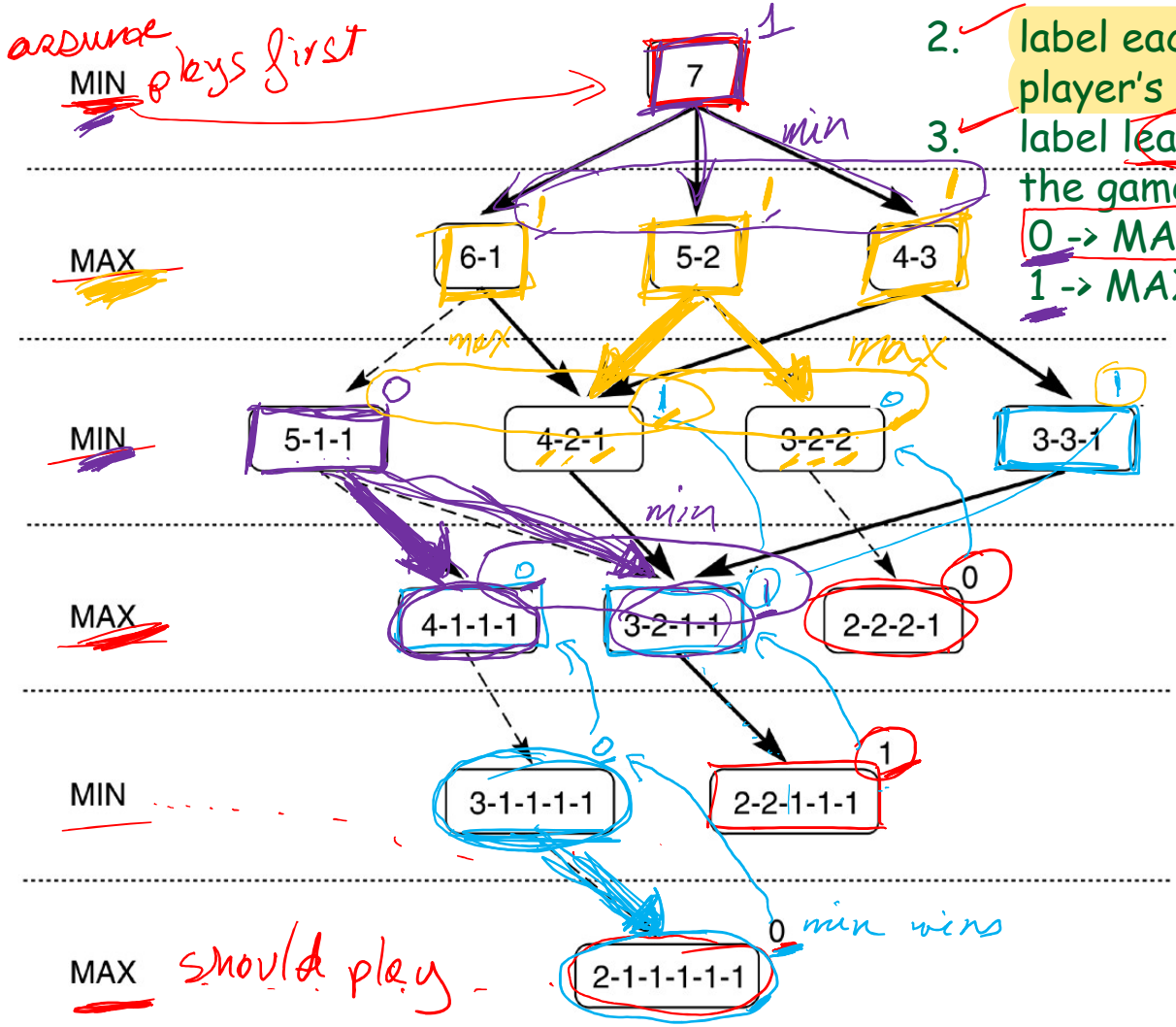


- eg. start with one pile of 7 tokens
- each step has to divide one pile into 2 non-empty piles of different sizes
- player without a move left loses game

1. build complete game tree

source: G. Luger (2005)

Exhaustive Minimax for Nim

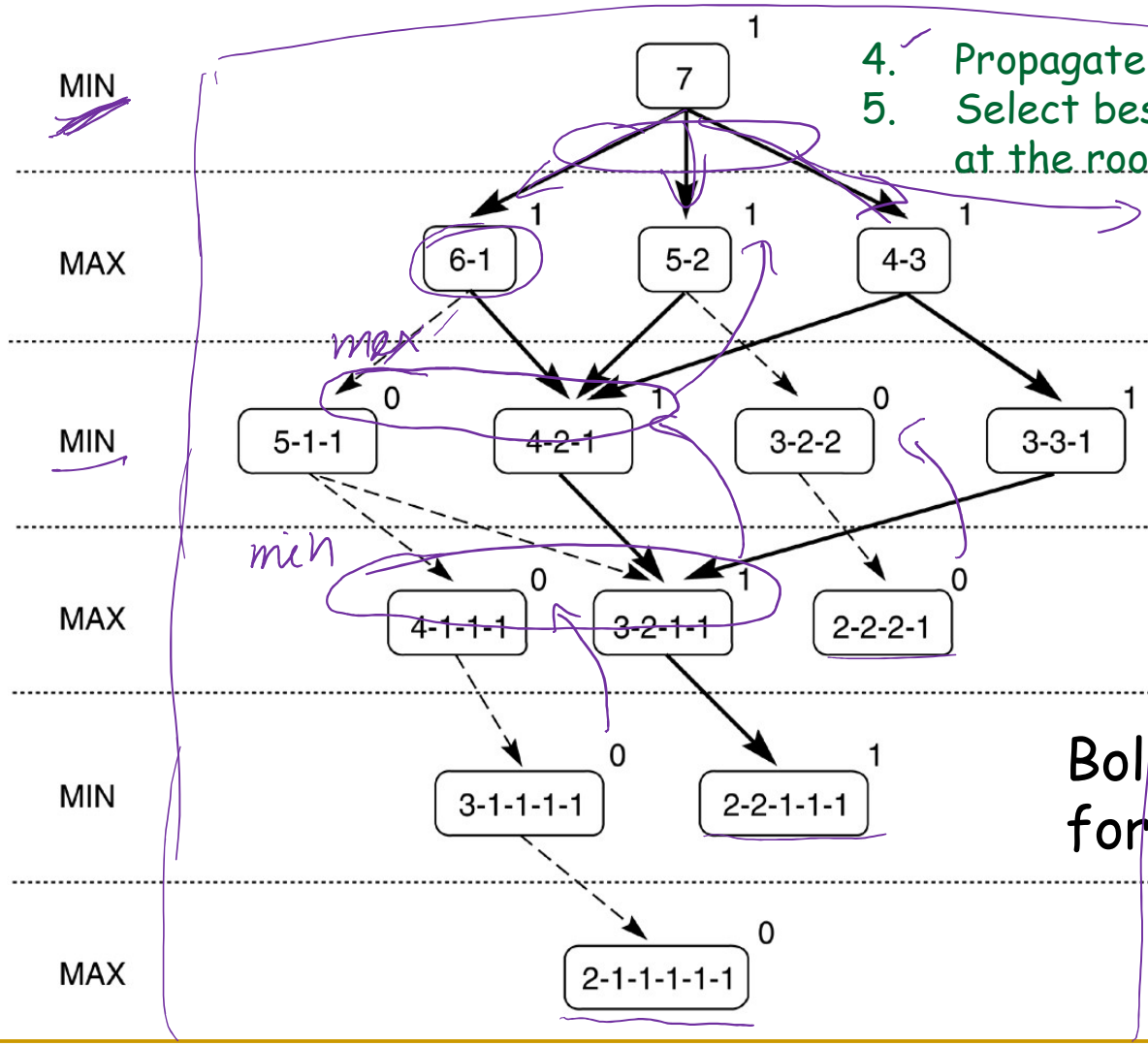


2. label each level according to player's turn (MAX or MIN)

3. label leaves with the outcome of the game
   0 -> MAX loses / MIN wins
   1 -> MAX wins / MIN loses

source: G. Luger (2005)

Exhaustive Minimax for Nim

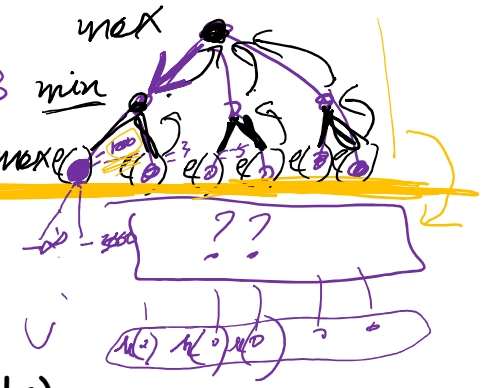


4. Propagate values up.
5. Select best next move for player at the root

Bold lines indicate forced win for MAX

source: G. Luger (2005)

n-ply Minimax with Heuristic



- problem with exhaustive Minimax...
  - state space for interesting games is too large!
- solution:
  - search only up to a predefined level
  - called n-ply look-ahead (n = max number of levels)
  - not an exhaustive search
  - nodes cannot be evaluation with win/loss/tie
  - nodes are evaluated with heuristics function e(n)
  - e(n) indicates how good a state seems to be for MAX compared to MIN
  - suffers from the horizon effect

https://www.wallpaperflare.com/beach-shore-sun-sunrise-sea-horizon-ship-water-sky-sunset-wallpaper-sbhsq

Heuristic Function for 2-player games

- simple strategy:
  - try to maximize difference between MAX's game and MIN's game
- typically called *e(n)*
- *e(n)* is a heuristic that estimates how favorable a node n is for MAX with respect to MIN
  - *e(n)* > 0 --> n is favorable to MAX
  - *e(n)* < 0 --> n is favorable to MIN
  - *e(n)* = 0 --> n is neutral

Choosing a Heuristic Function e(n)

- Usually e(n) is a weighted sum of various features:

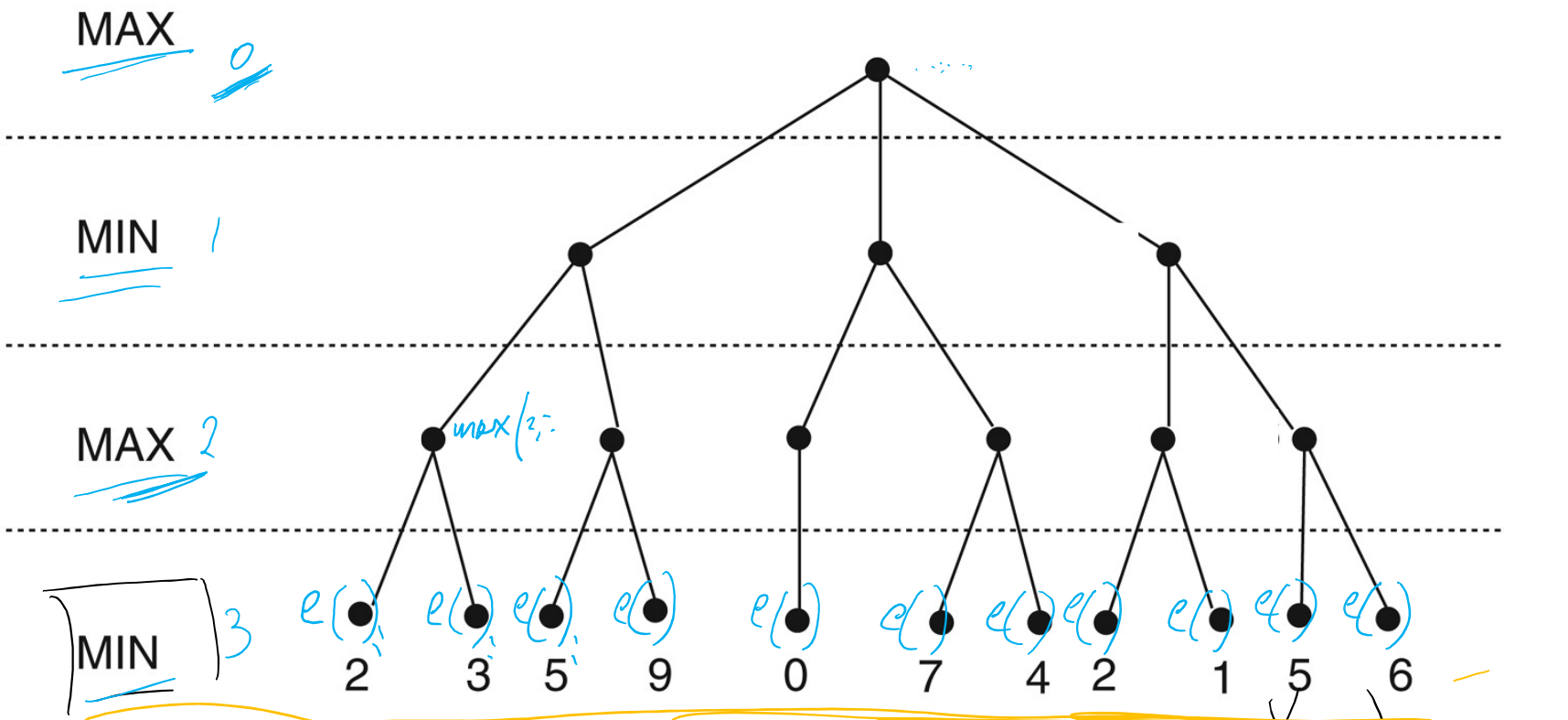
$$e(n) = \sum w_i f_i(n)$$

dependent on actual game

- E.g. of features:
  - f_1 = number of pieces left on the game for MAX
  - f_2 = number of possible moves left for MAX
  - f_3 = -(number of pieces left on the game for MIN)
  - f_4 = -(number of possible moves left for MIN)
- E.g. of weights:
  - w_1 = 0.5 // f_1 is a very important feature
  - w_2 = 0.2 // f_2 is not very important
  - w_3 = 0.2 // f_3 is not very important
  - w_4 = 0.1 // f_4 is really not important

f_5: queen still alive (1/0)
f_6: # of bishops left
$w_5 = 0.3$

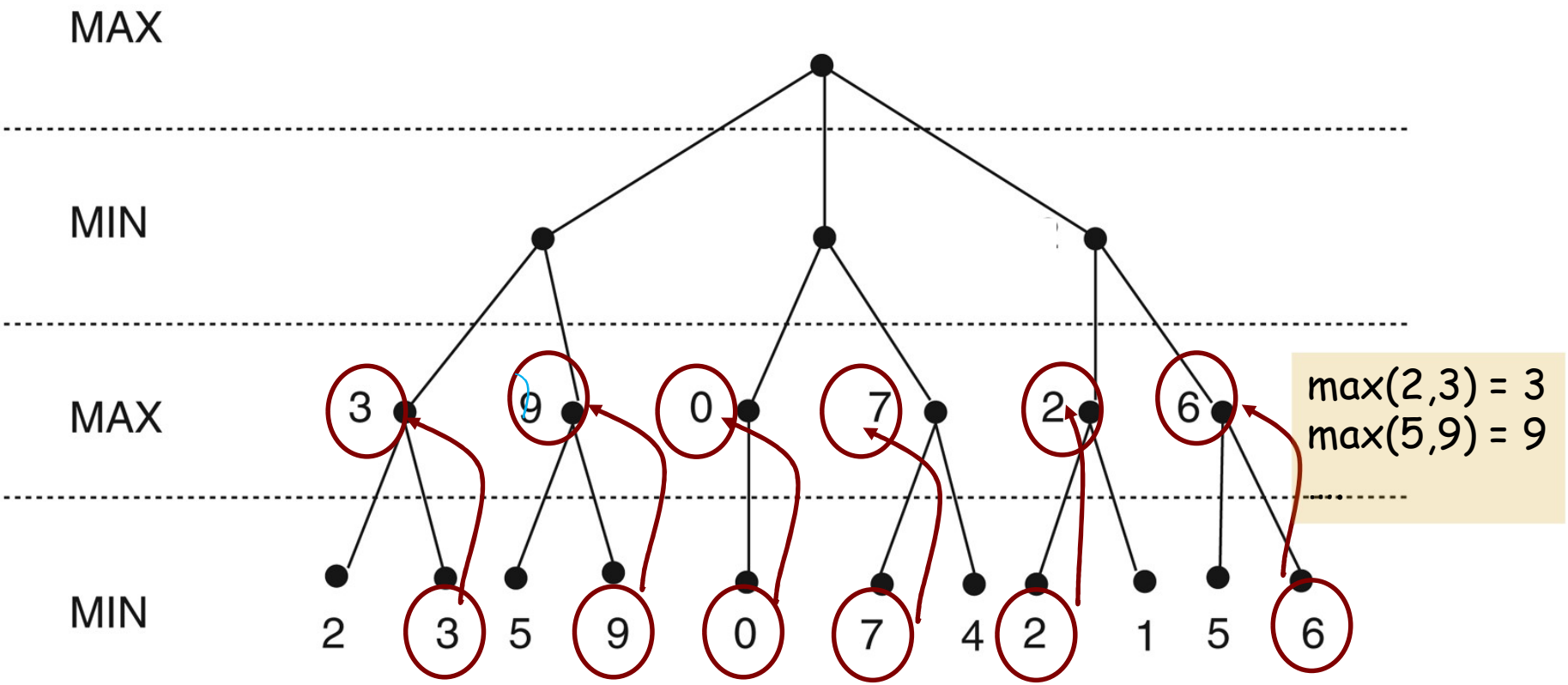
Minimax with 3-ply look-ahead



Leaf nodes show the actual heuristic value e(n)

source: G. Luger (2005)

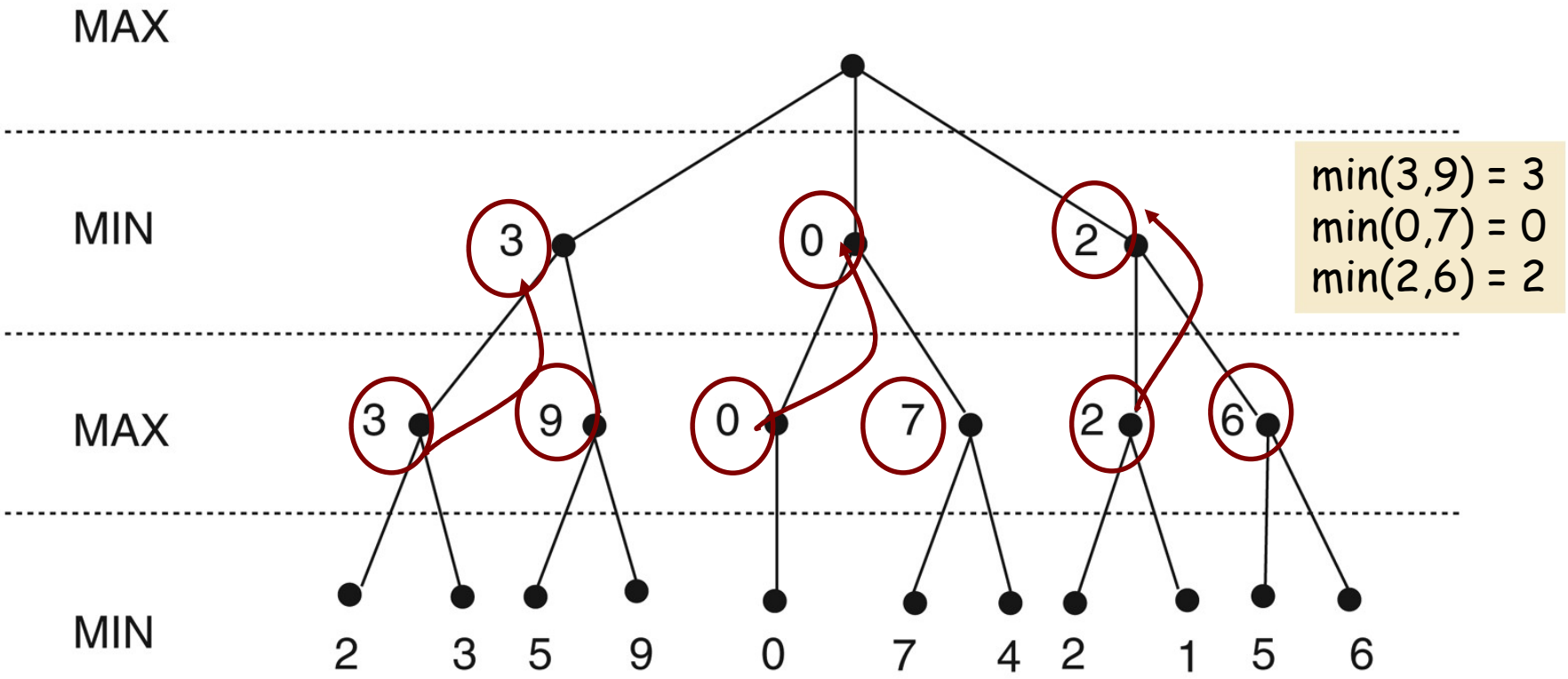
Minimax with 3-ply look-ahead



Leaf nodes show the actual heuristic value *e(n)*
Internal nodes show back-up heuristic value

source: G. Luger (2005)

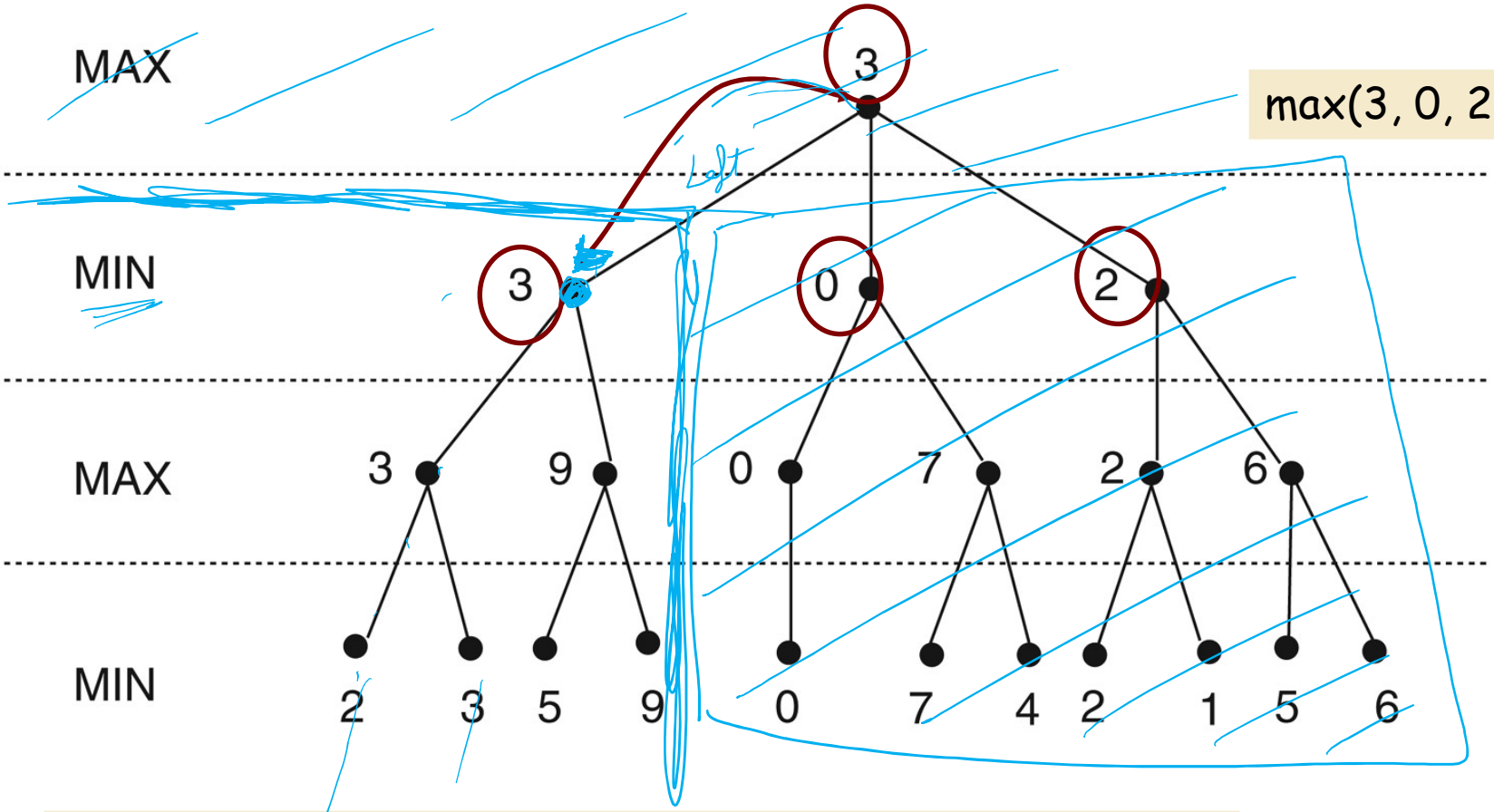
Minimax with 3-ply look-ahead



Leaf nodes show the actual heuristic value *e(n)*
Internal nodes show back-up heuristic value

source: G. Luger (2005)

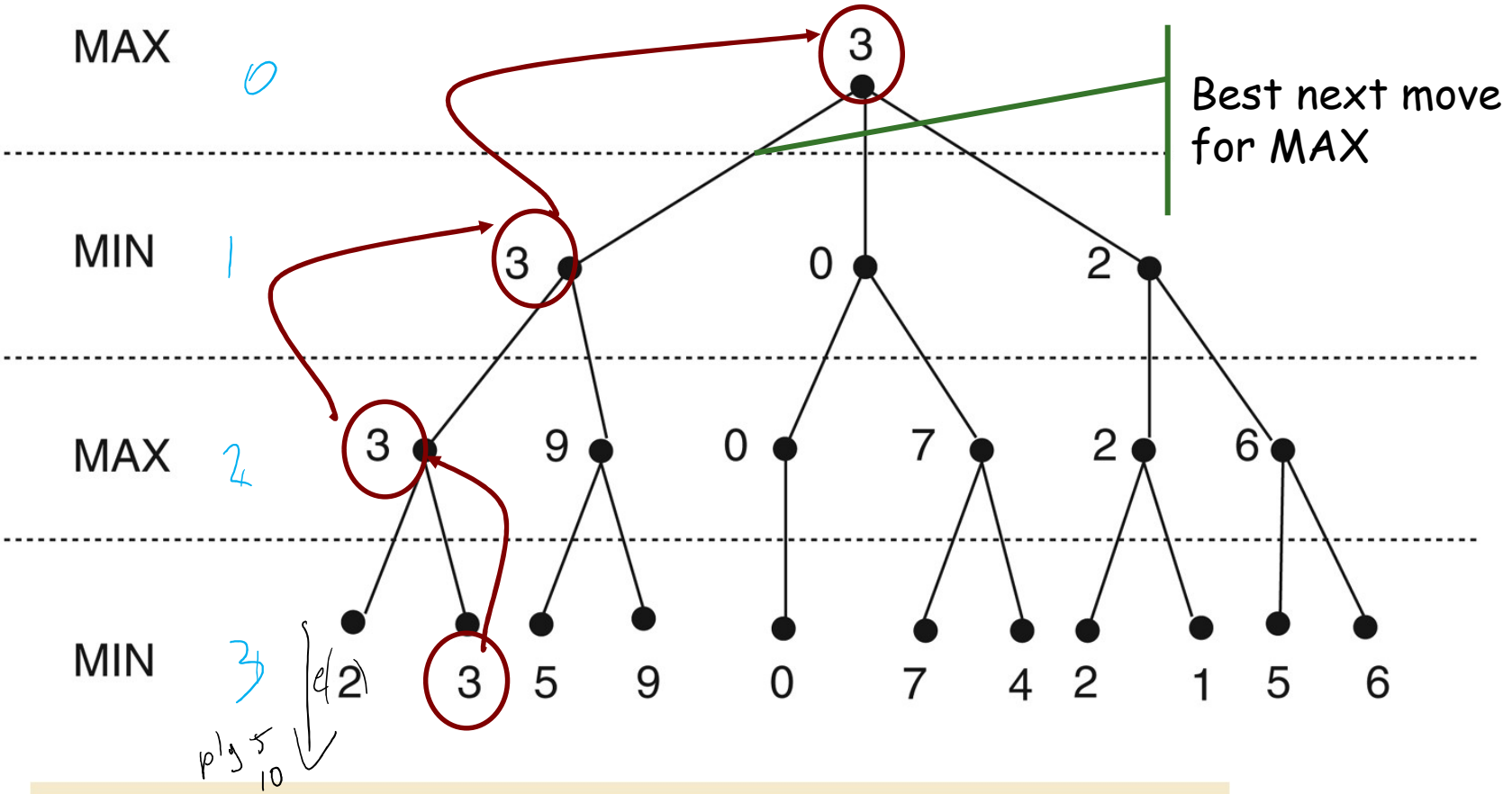
Minimax with 3-ply look-ahead



Leaf nodes show the actual heuristic value e(n)
Internal nodes show back-up heuristic value

source: G. Luger (2005)

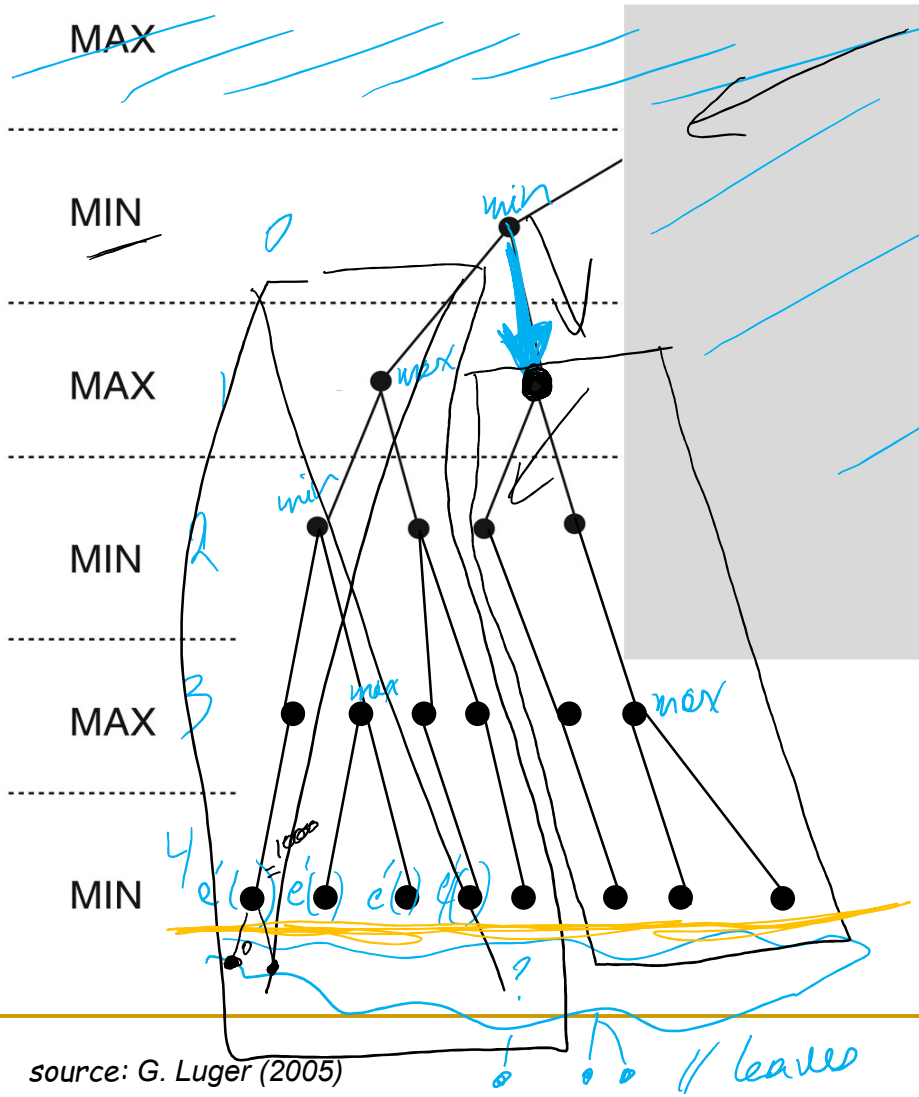
Minimax with 3-ply look-ahead



Leaf nodes show the actual heuristic value *e(n)*
Internal nodes show back-up heuristic value

source: G. Luger (2005)

and then, MIN will play

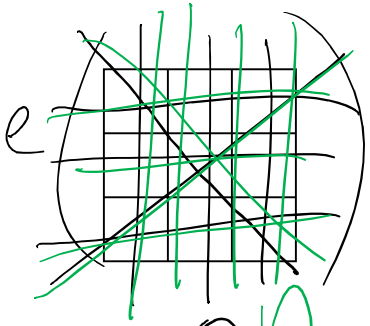


source: G. Luger (2005)

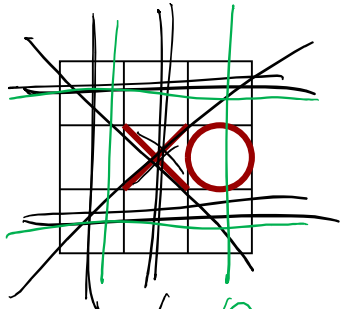
Example: *e(n)* for Tic-Tac-Toe

- assume MAX plays X
- possible *e(n)*

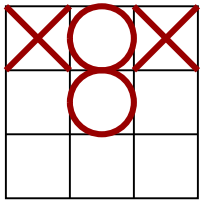
$$e(n) = \begin{cases} \text{number of rows, columns, and diagonals open for MAX} \\ \quad - \text{number of rows, columns, and diagonals open for MIN} \\ +\infty, \text{ if n is a forced win for MAX} \\ -\infty, \text{ if n is a forced win for MIN} \end{cases}$$



$e(n) = 8 - 8 = 0$

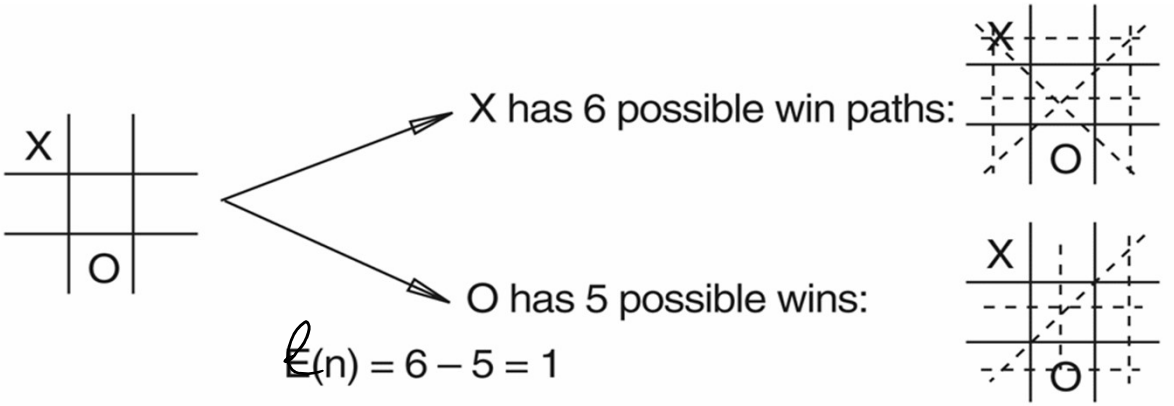


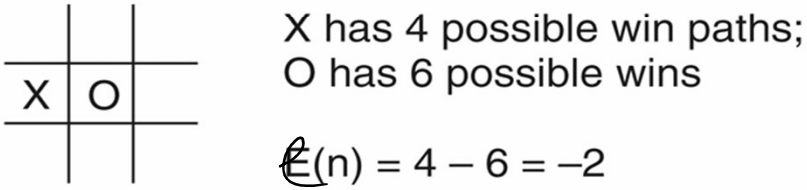
$e(n) = 6 - 4 = 2$

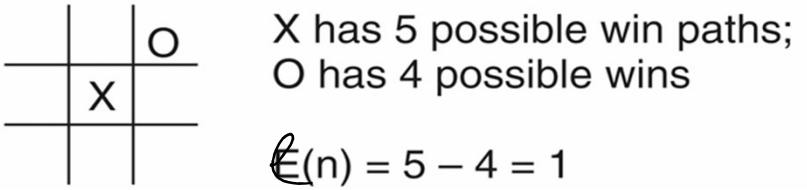


$e(n) = 3 - 3 = 0$

More examples…

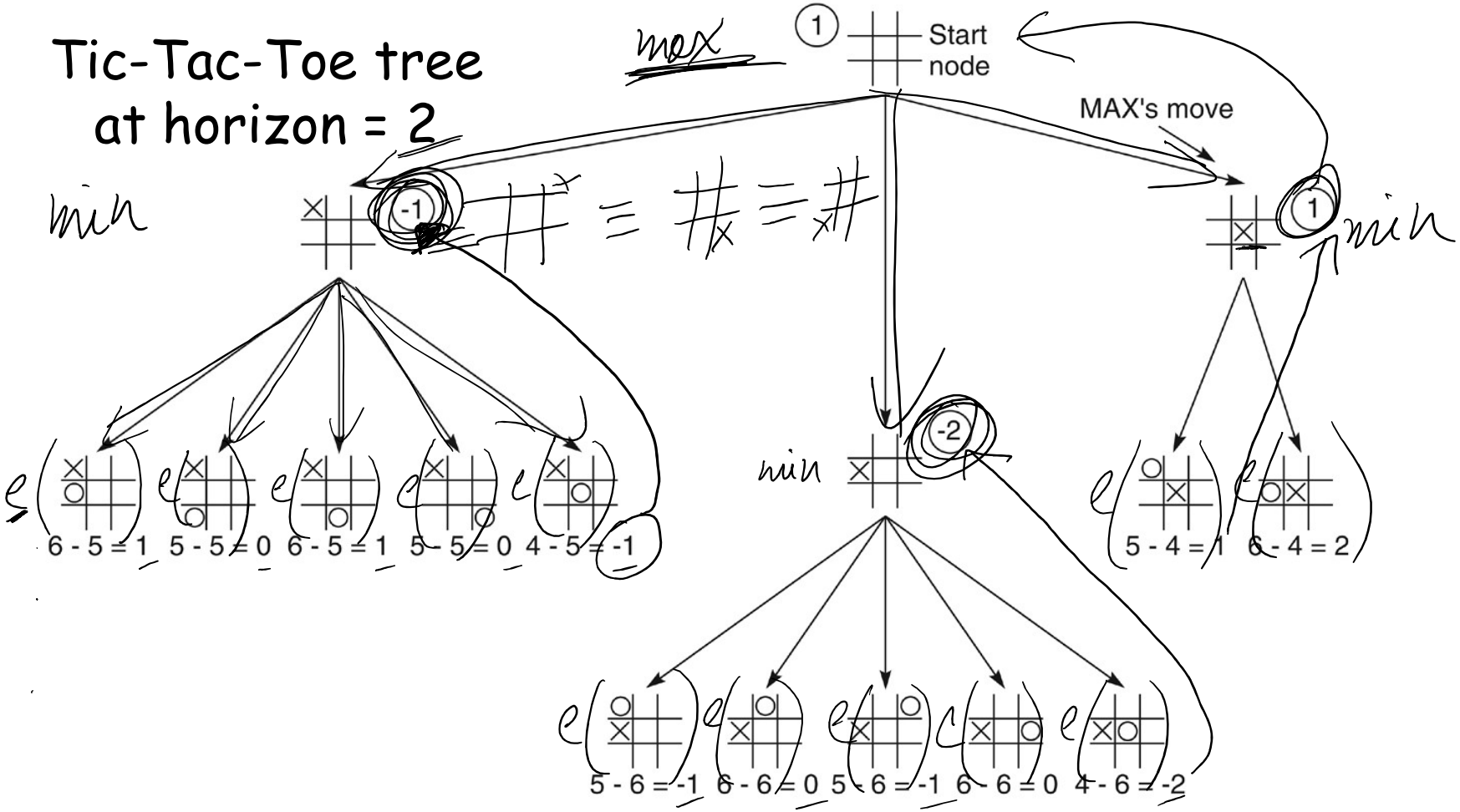






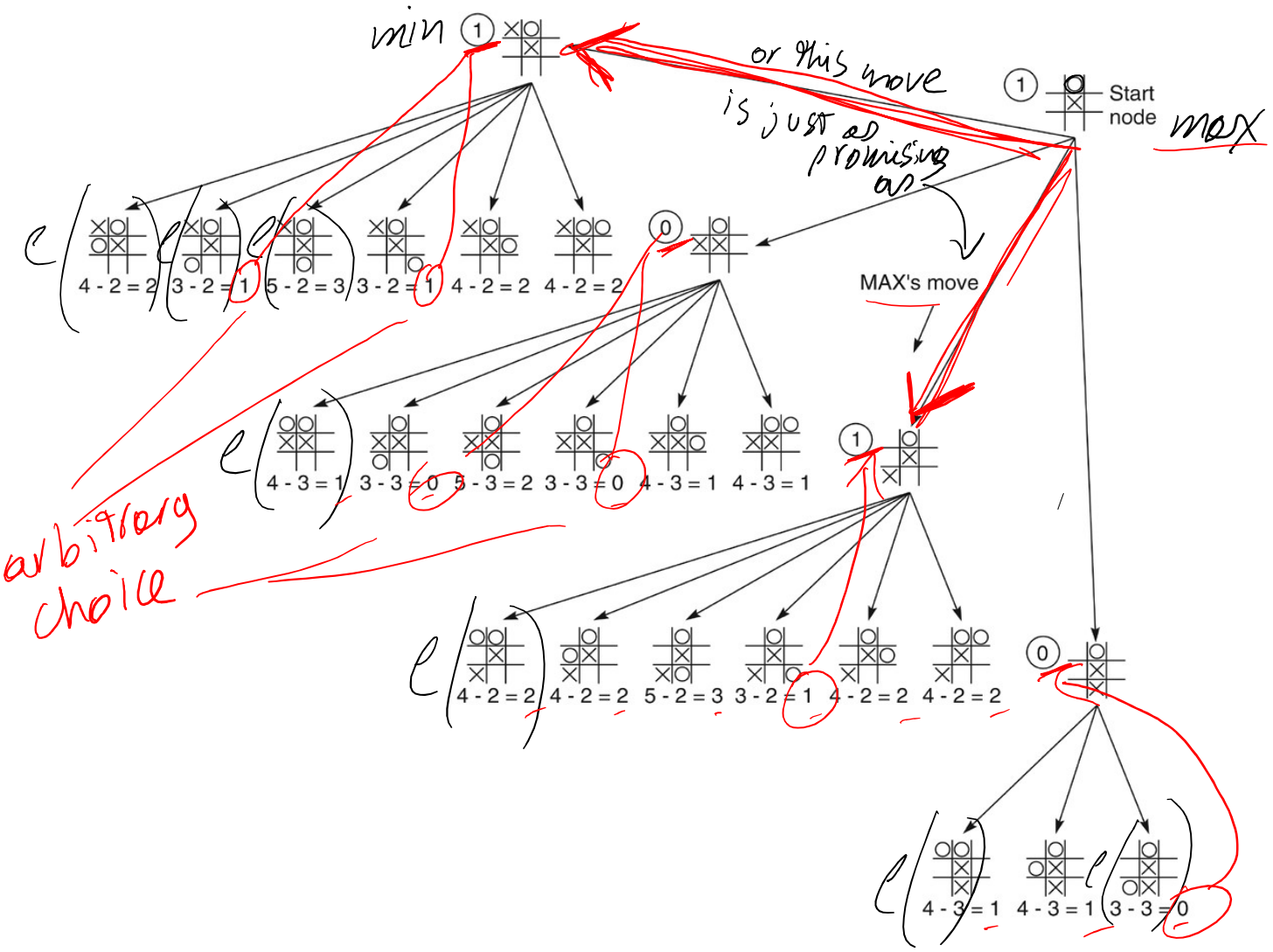
source: G. Luger (2005)

2-ply Minimax for Opening Move



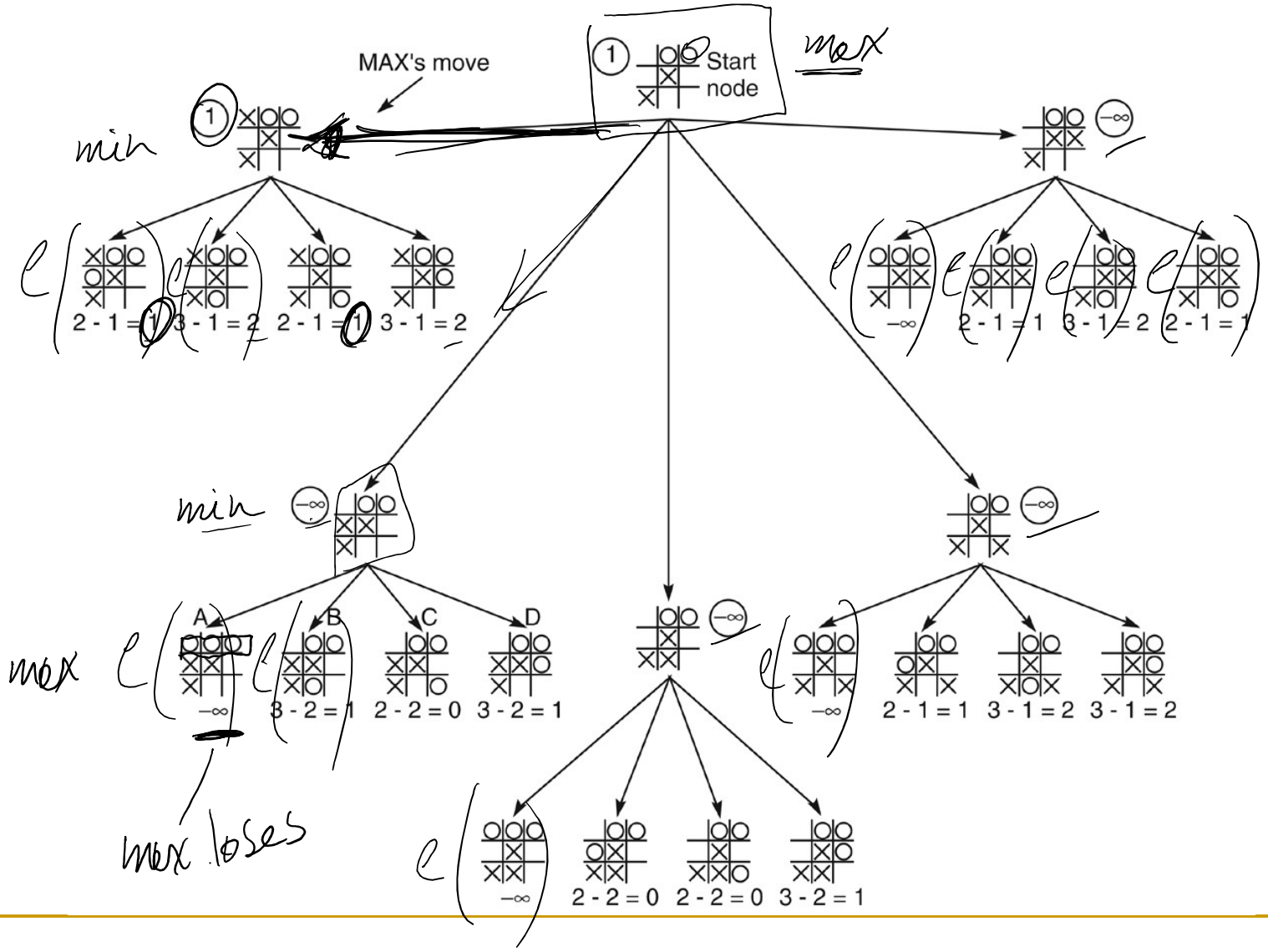
source: G. Luger (2005)

2-ply Minimax: MAX's possible 2nd moves



source: G. Luger (2005)

2-ply Minimax: MAX's move at end



source: G. Luger (2005)

Today

- Adversarial Search ✓
  1. Minimax ✓
  2. Alpha-Beta Pruning
  3. Other Adversarial Search
     1. Multiplayer Games
     2. Stochastic Games
     3. Monte Carlo Tree Search

Up Next

- Adversarial Search
  1. Minimax
  2. Alpha-Beta Pruning
  3. Other Adversarial Search
     1. Multiplayer Games
     2. Stochastic Games
     3. Monte Carlo Tree Search

COMP 472 Artificial Intelligence: Adversarial Search part 4
Alpha-Beta Pruning video 2

- Russell & Norvig: Chapter 5

Today

- Adversarial Search
  1. Minimax
  2. Alpha-beta pruning 

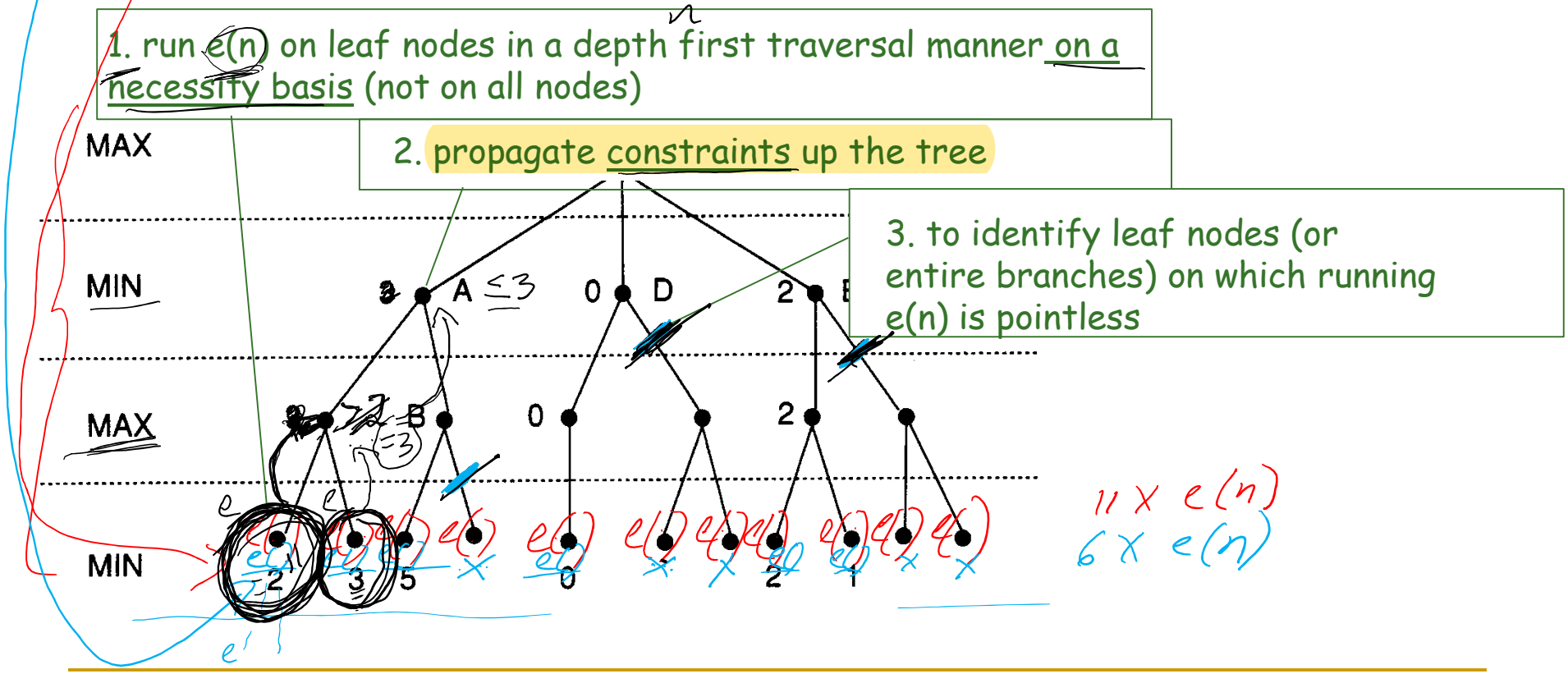
  3. Other Adversarial Searc.
     1. Multiplayer Games
     2. Stochastic Games
     3. Monte Carlo Tree Search

Alpha-Beta Pruning

- Optimization over Minimax, that:
  - ignores (cuts off/prunes) branches of the tree that cannot contribute to the solution
  - reduces branching factor
  - allows deeper search with same effort

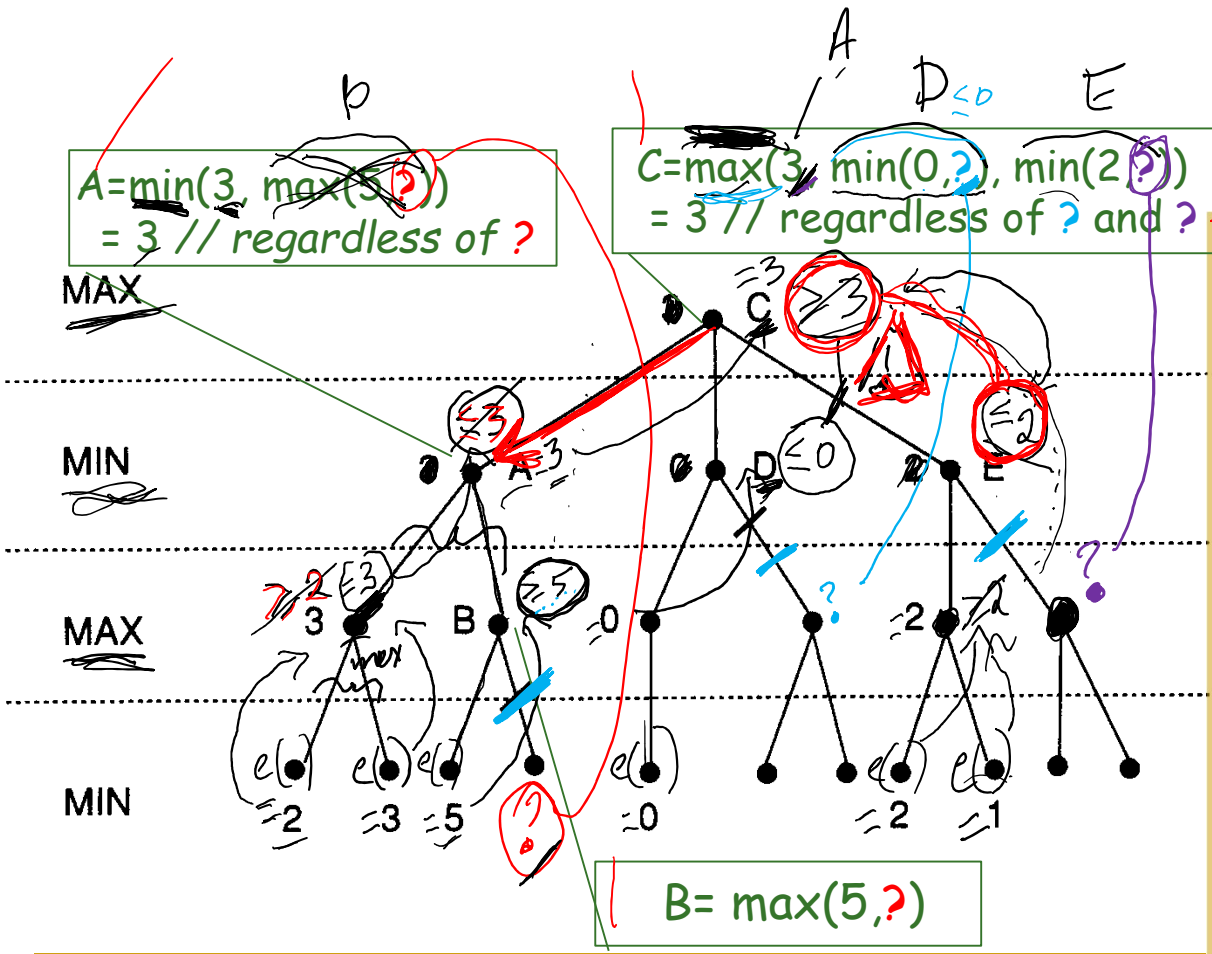
Alpha-Beta Pruning: Example 1

- With Minimax, we look at all nodes at depth *n*
- With α-β pruning, we ignore branches that could not possibly contribute to the final decision



source: G. Luger (2005)

Alpha-Beta Pruning: Example 1



B will be >= 5

So we can ignore B's right branch, because A must be 3

D will be <= 0

But C will be >= 3

So we can ignore D's right branch

E will be <= 2.

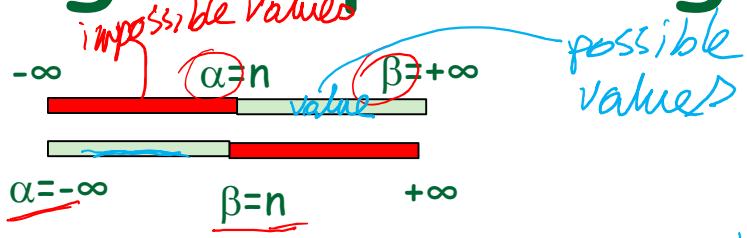
So we can ignore E's right branch

Because C will be 3.

source: G. Luger (2005)

Alpha-Beta Pruning – α pruning

- α = minimum possible value value ≥ n
- β = maximum possible value value ≤ n

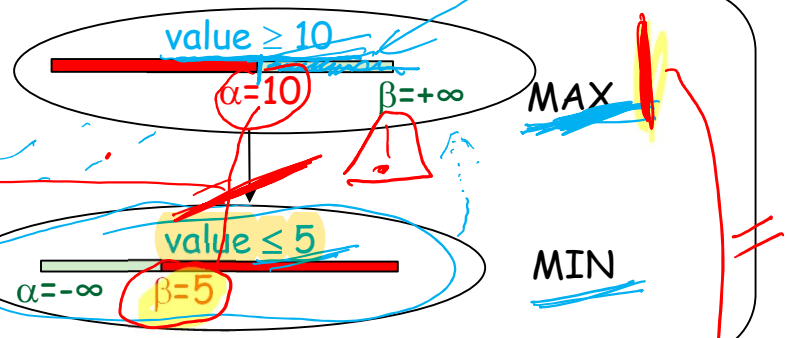


- Alpha pruning – parent is a MAX node

1. if MAX node's α = n, then we can prune branches from a MIN descendant that has a β ≤ n.

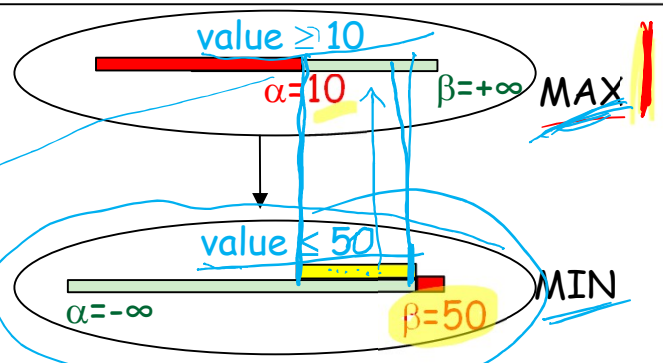
   if β_child ≤ α_ancestor → prune

   incompatible...
   so stop searching this branch;
   the value cannot come from there!



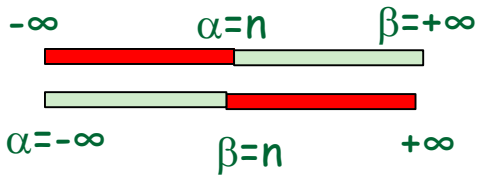
2. if β_child > α_ancestor → cannot prune

   compatible...
   we need to search this branch;
   the value could come from there!



Alpha-Beta Pruning - β pruning

- α = minimum possible value — value ≥ n
- β = maximum possible value — value ≤ n

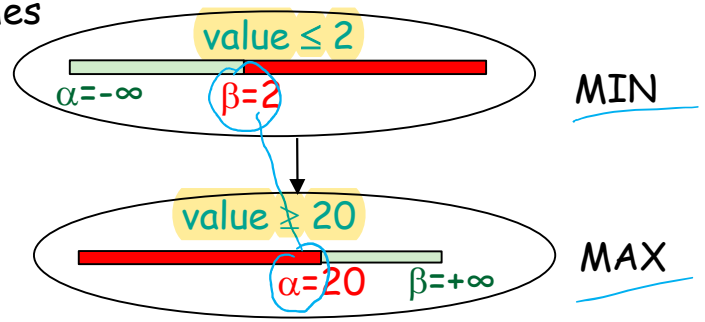


same

- Beta pruning – parent is a MIN node

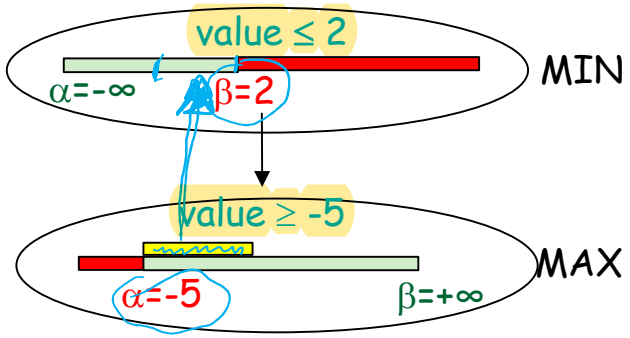
1. if a MIN node's β = n, then we can prune branches from a MAX descendant that has an α ≥ n.
   if α_child ≥ β _ancestor → prune

incompatible...
so stop searching this branch;
the value cannot come from there!



2. if α_child < β _ancestor → cannot prune

compatible...
we need to search this branch;
the value could come from there!



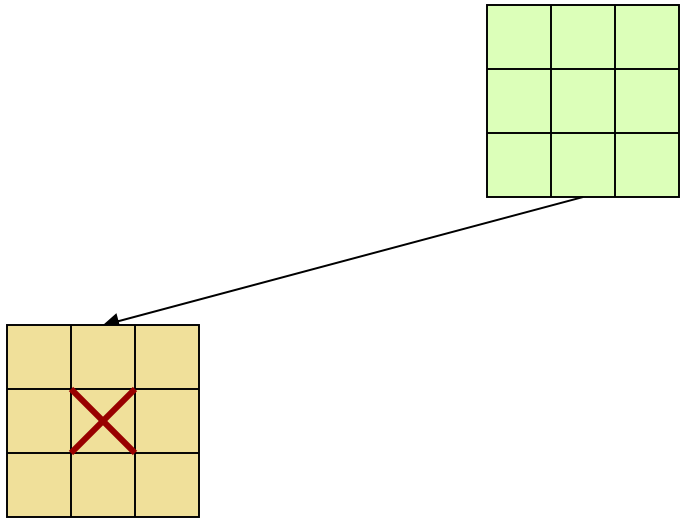
Alpha-Beta Pruning Algorithm

```
01 function alphabeta(node, depth, α, β, maximizingPlayer)
02      if depth = 0 or node is a terminal node
03          return the heuristic value of node
04      if maximizingPlayer
05          v := -∞
06          for each child of node
07              v := max(v, alphabeta(child, depth - 1, α, β, FALSE))
08              α := max(α, v)
09              if β ≤ α
10                  break (* β cut-off *)
11          return v
12      else
13          v := ∞
14          for each child of node
15              v := min(v, alphabeta(child, depth - 1, α, β, TRUE))
16              β := min(β, v)
17              if β ≤ α
18                  break (* α cut-off *)
19          return v
```

Initial call:
`alphabeta(origin, depth, -∞, +∞, TRUE)`

source: http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Example with tic-tac-toe

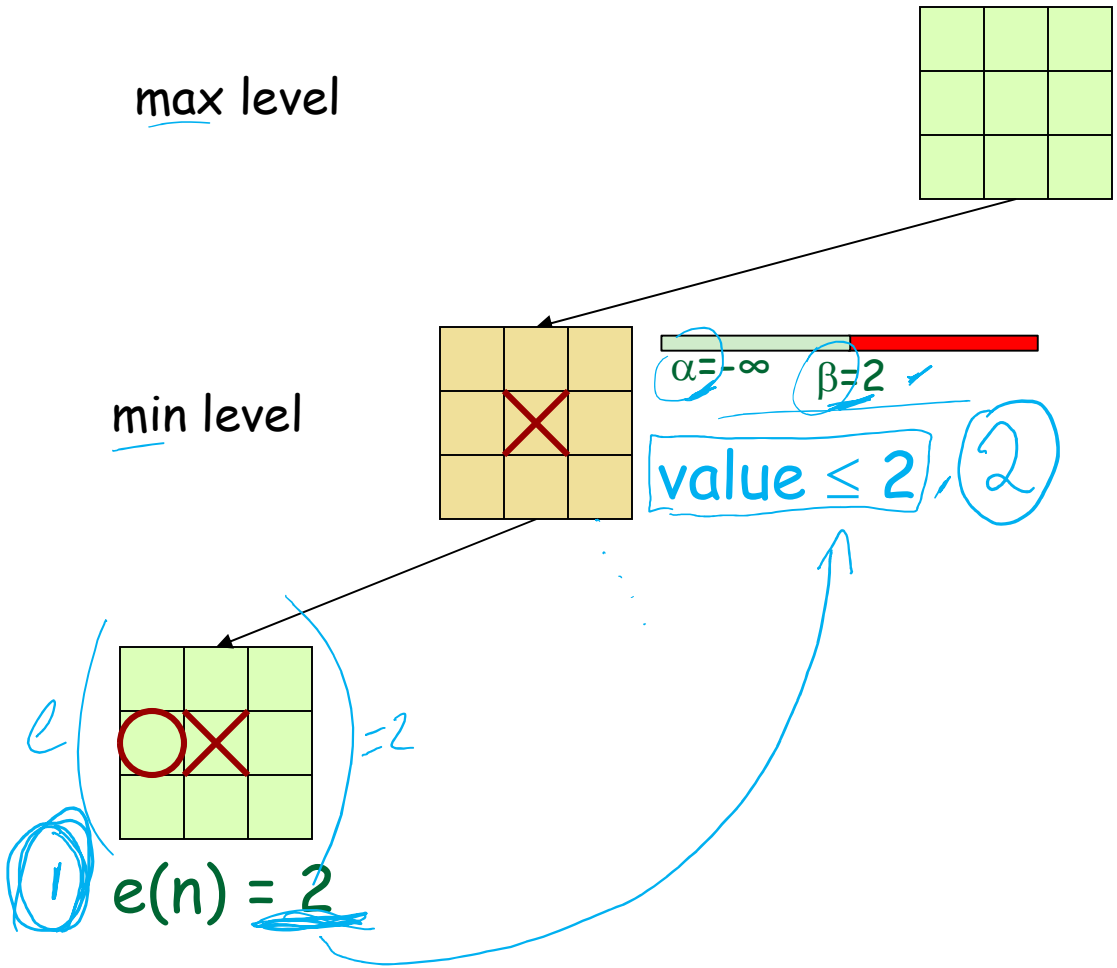


max level

min level

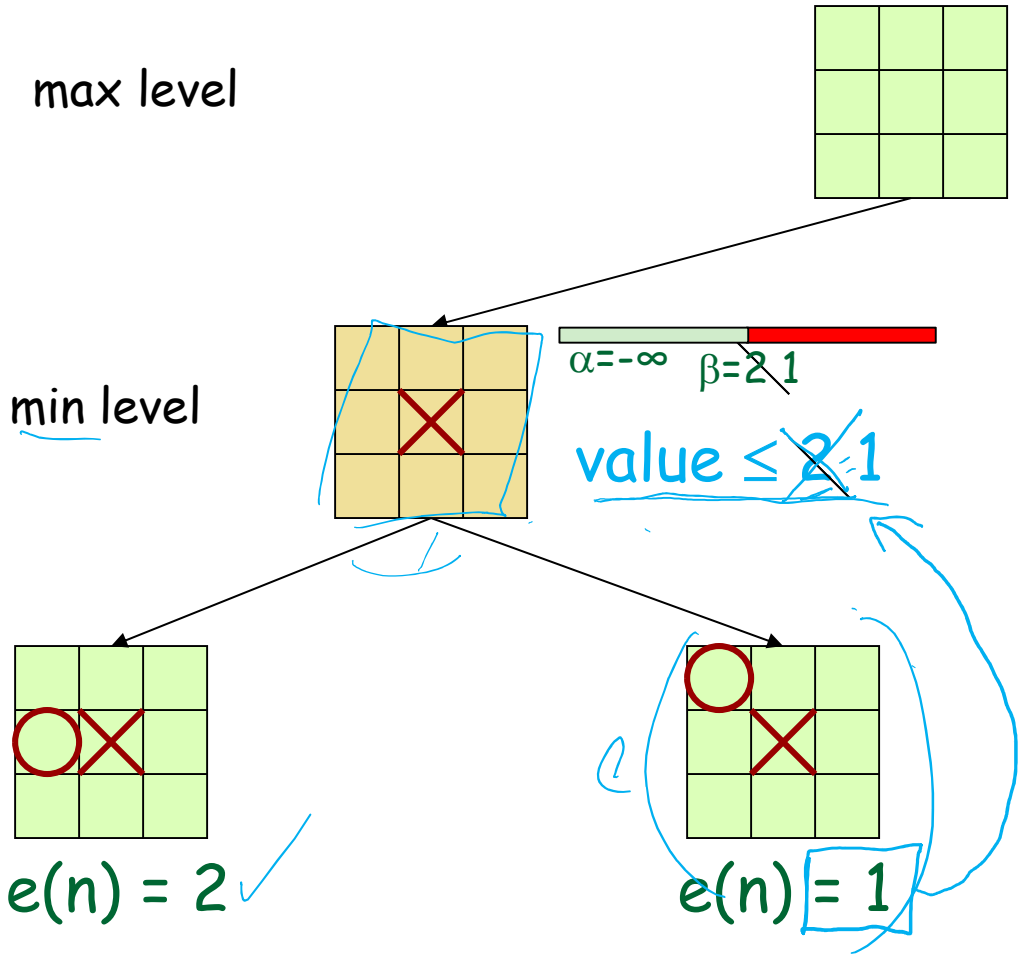
source: robotics.stanford.edu/~latombe/cs121/2003/home.htm

Example with tic-tac-toe



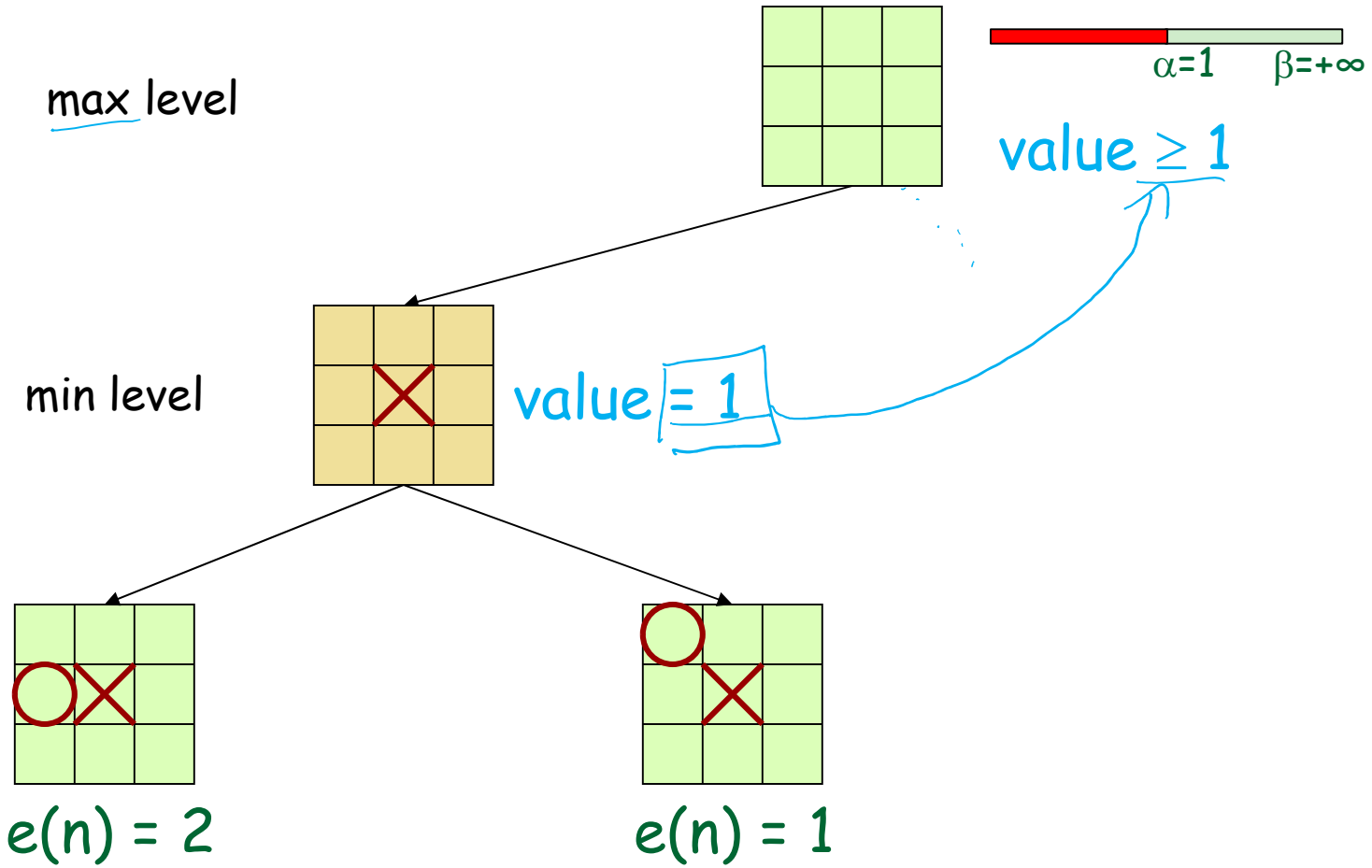
source: robotics.stanford.edu/~latombe/cs121/2003/home.htm

Example with tic-tac-toe



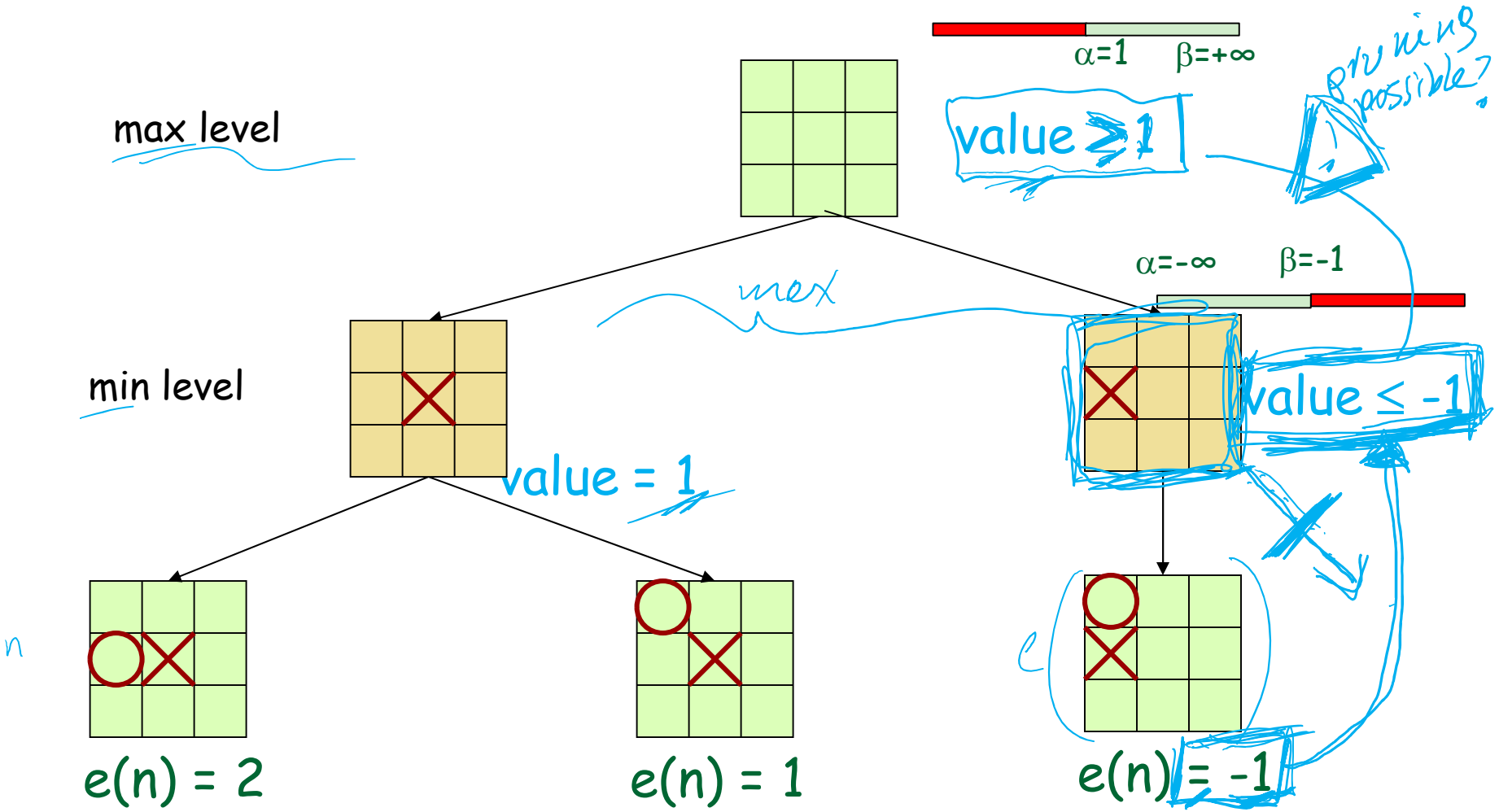
source: robotics.stanford.edu/~latombe/cs121/2003/home.htm

Example with tic-tac-toe



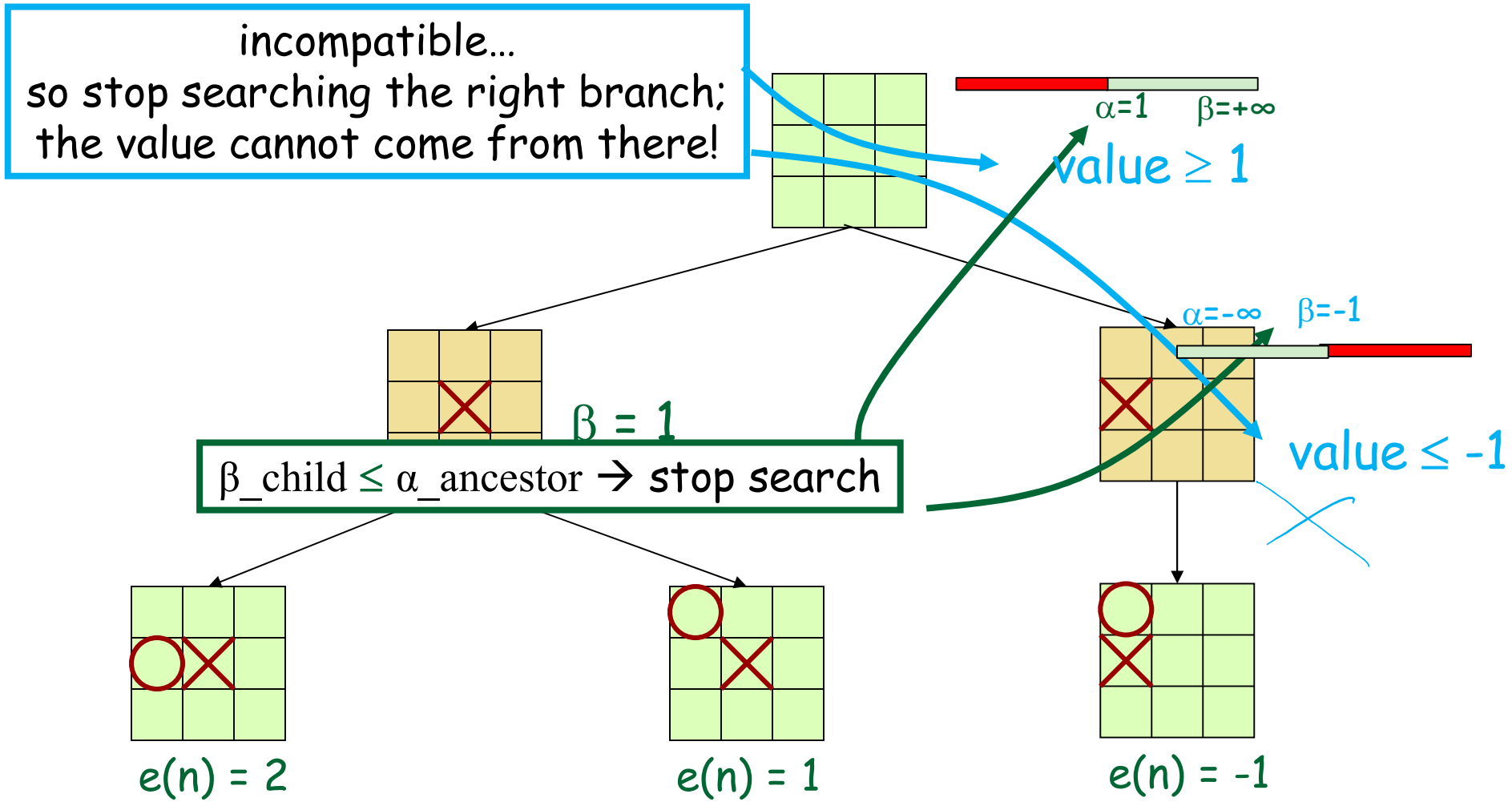
source: robotics.stanford.edu/~latombe/cs121/2003/home.htm

Example with tic-tac-toe



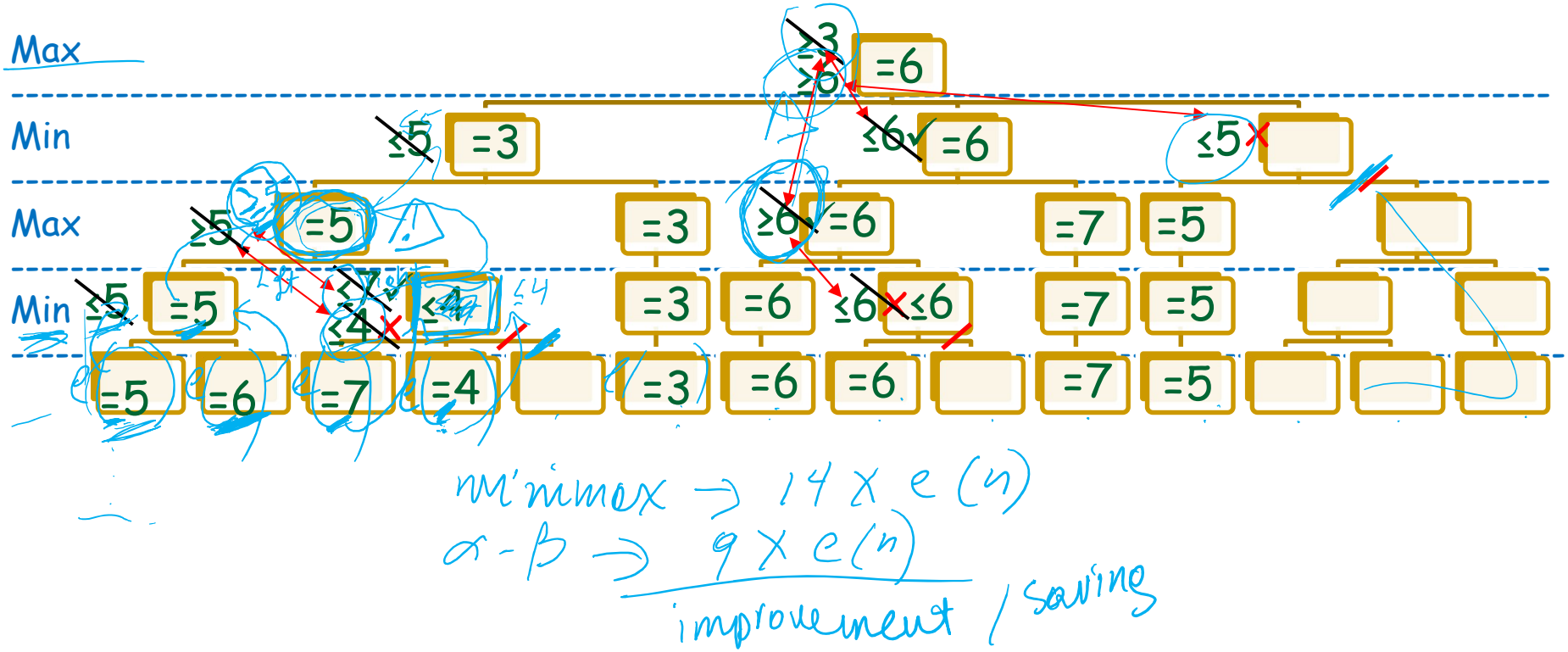
source: robotics.stanford.edu/~latombe/cs121/2003/home.htm

Example with tic-tac-toe



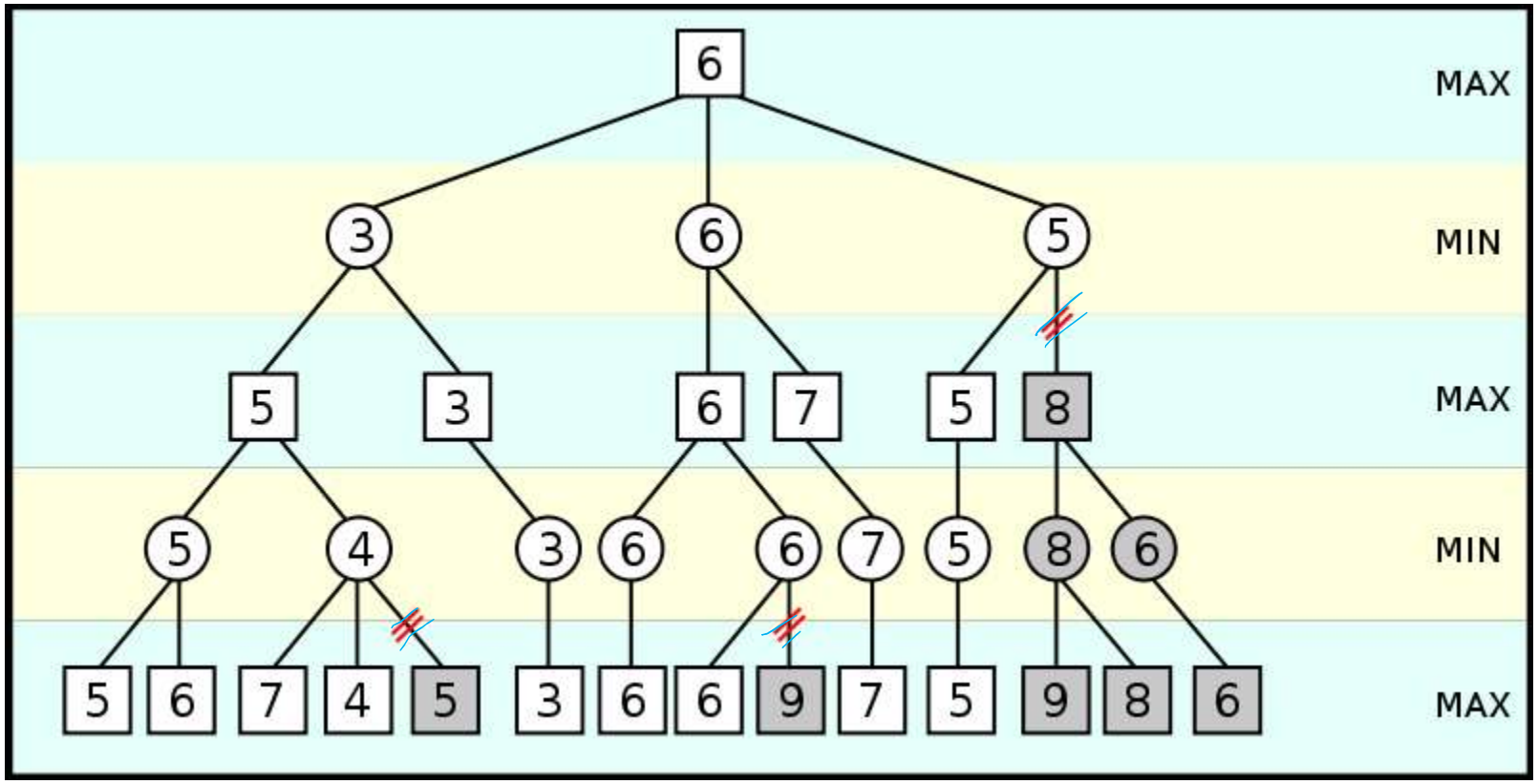
source: robotics.stanford.edu/~latombe/cs121/2003/home.htm

Alpha-Beta Pruning: Example 2



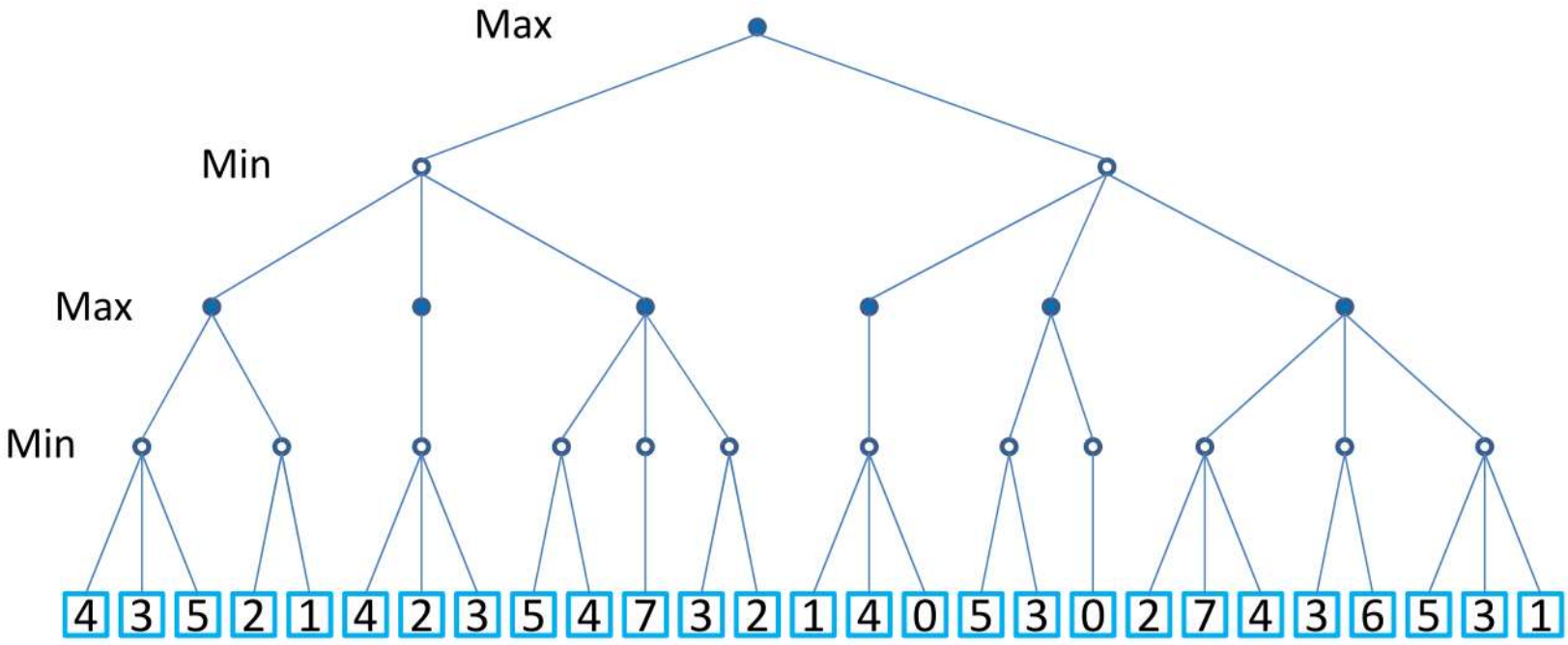
source: http://en.wikipedia.org/wiki/File:AB_pruning.svg

Alpha-Beta Pruning: Example 2

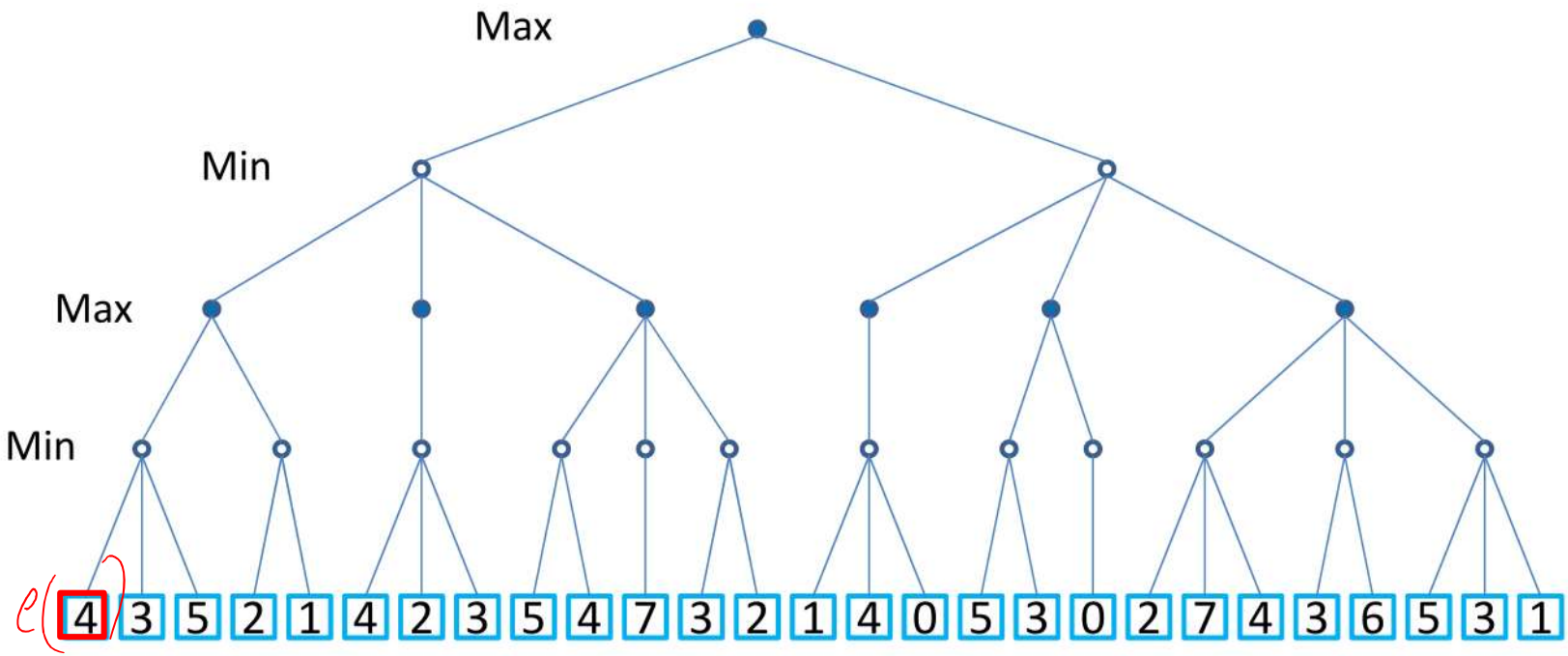


source: http://en.wikipedia.org/wiki/File:AB_pruning.svg

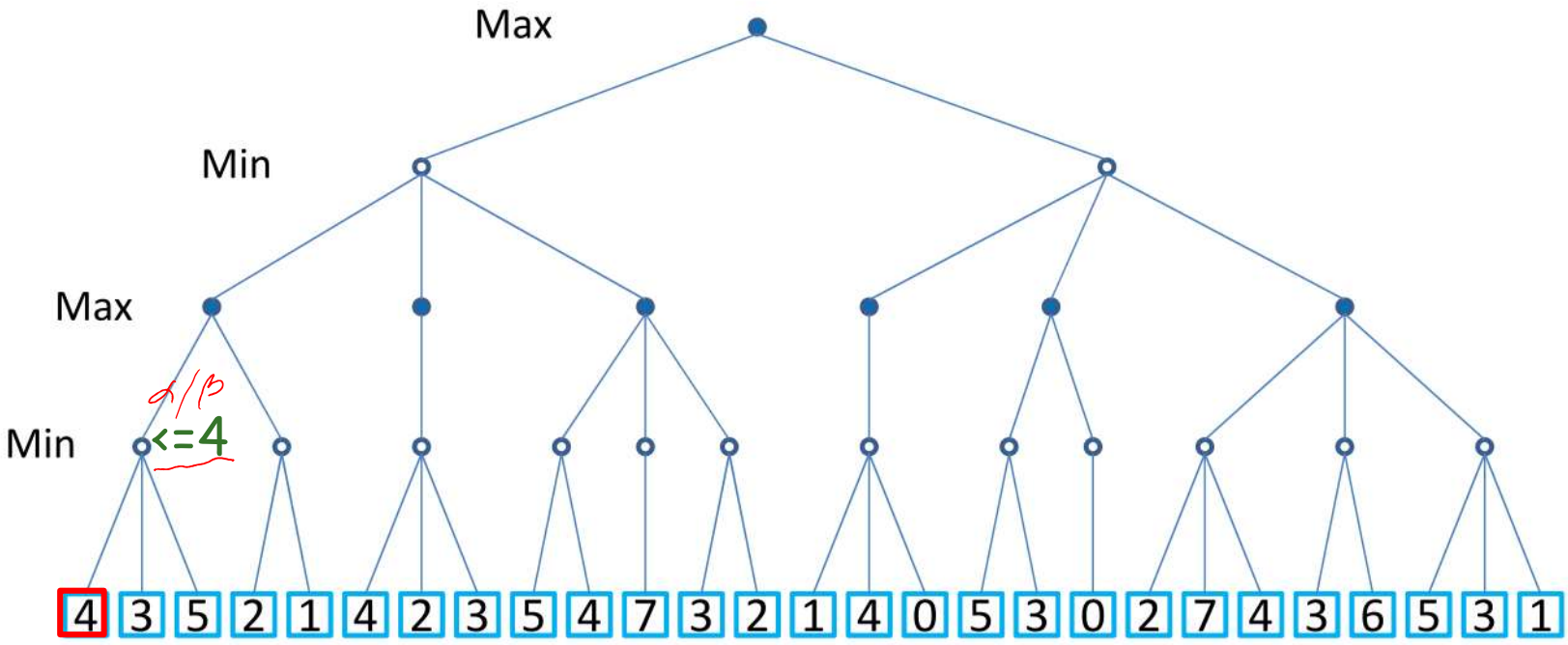
Alpha-Beta Pruning: Example 3



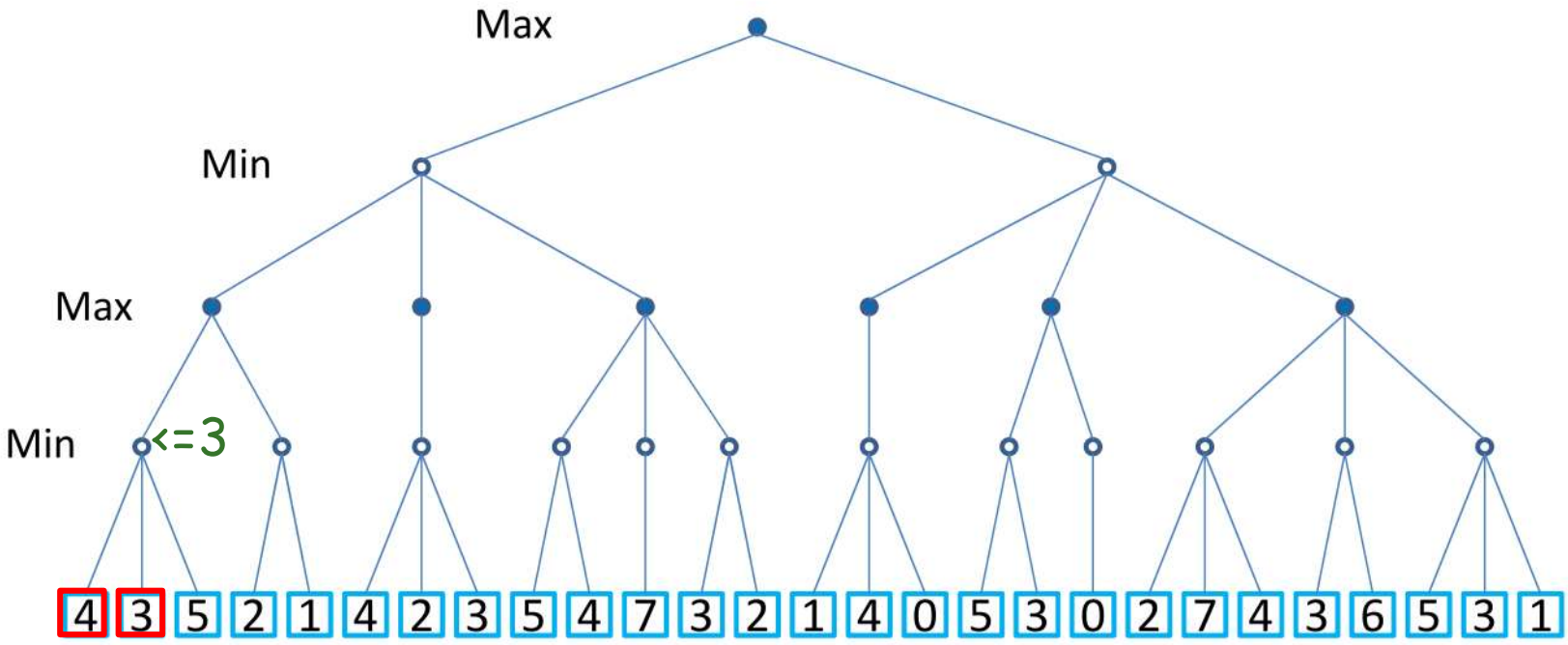
Alpha-Beta Pruning: Example 3



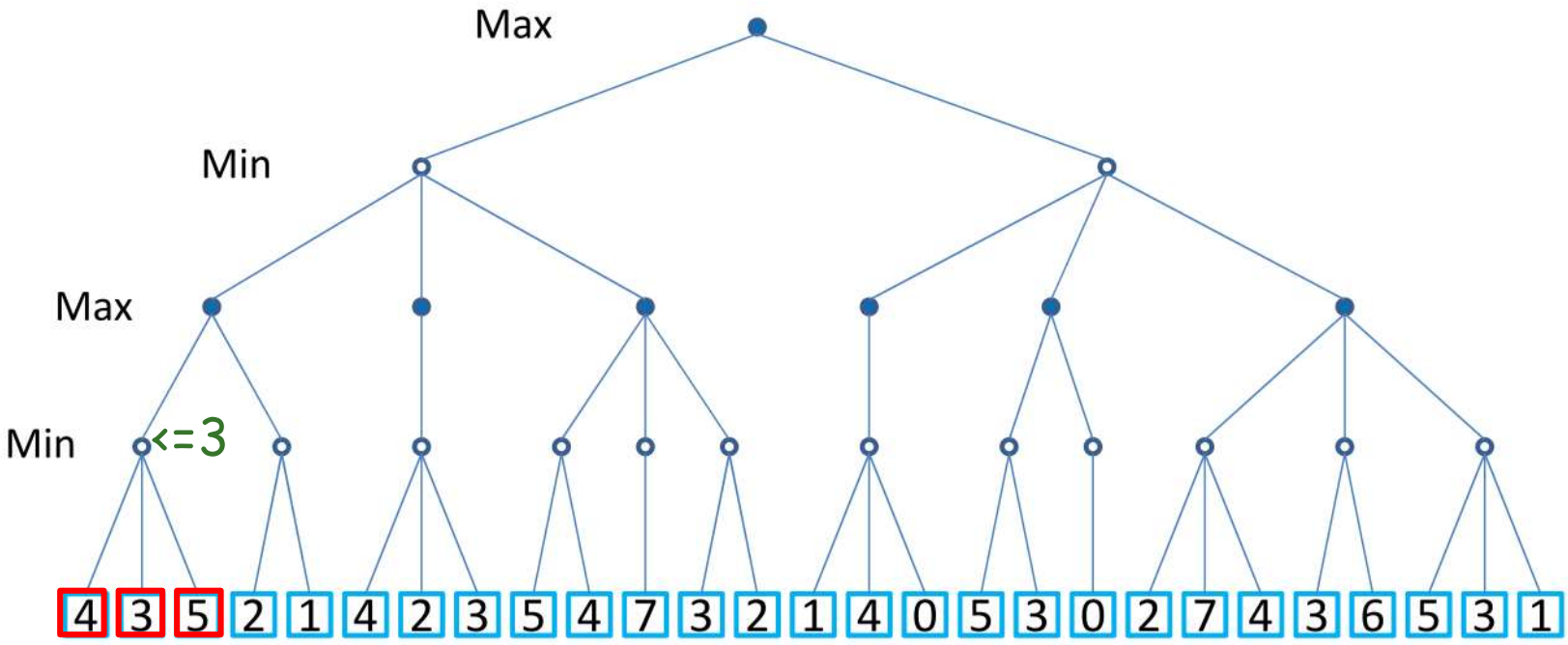
Alpha-Beta Pruning: Example 3



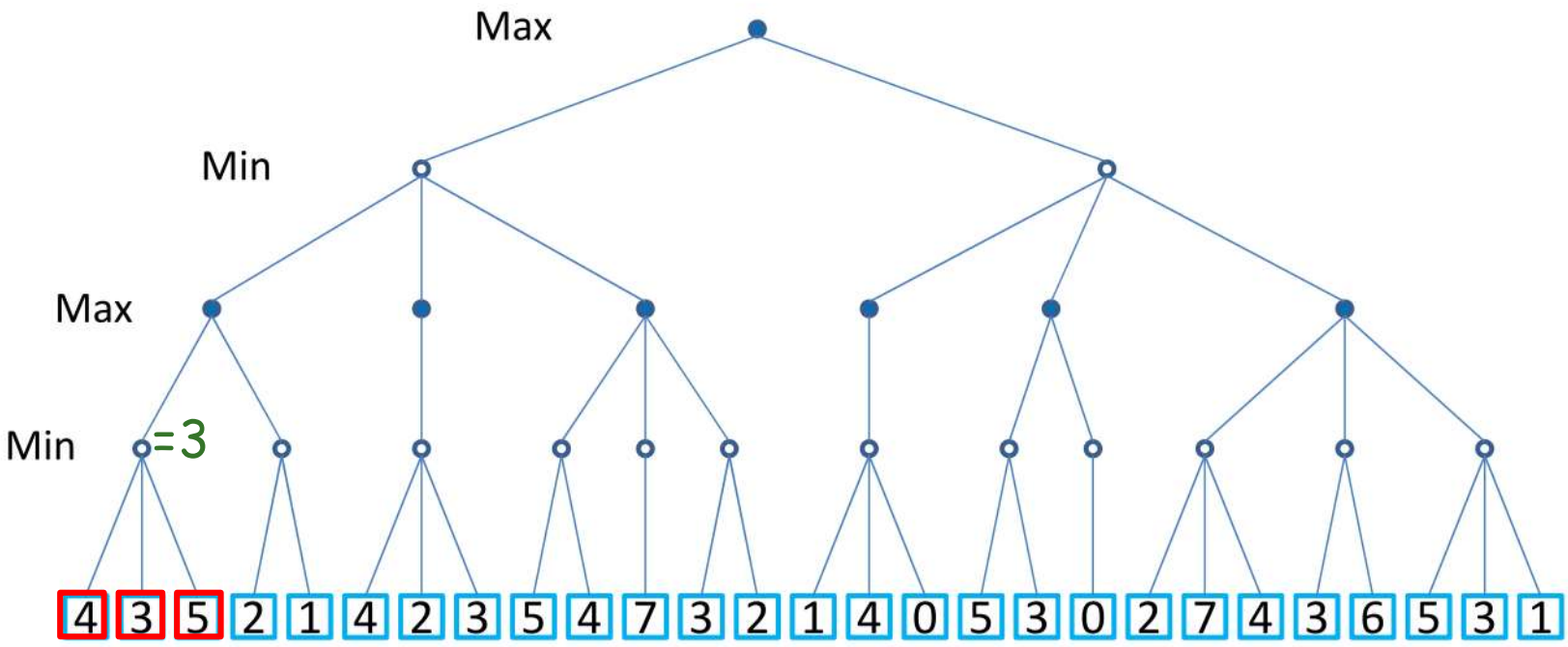
Alpha-Beta Pruning: Example 3



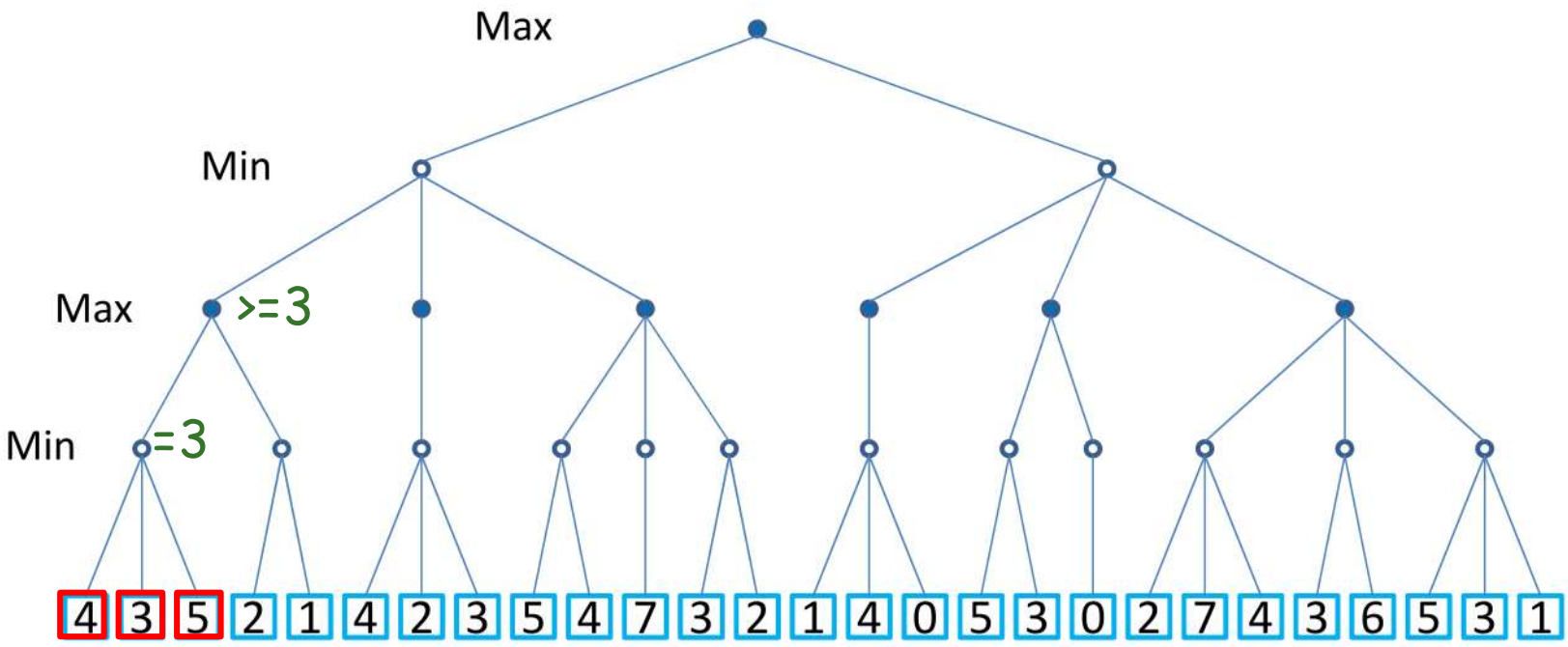
Alpha-Beta Pruning: Example 3



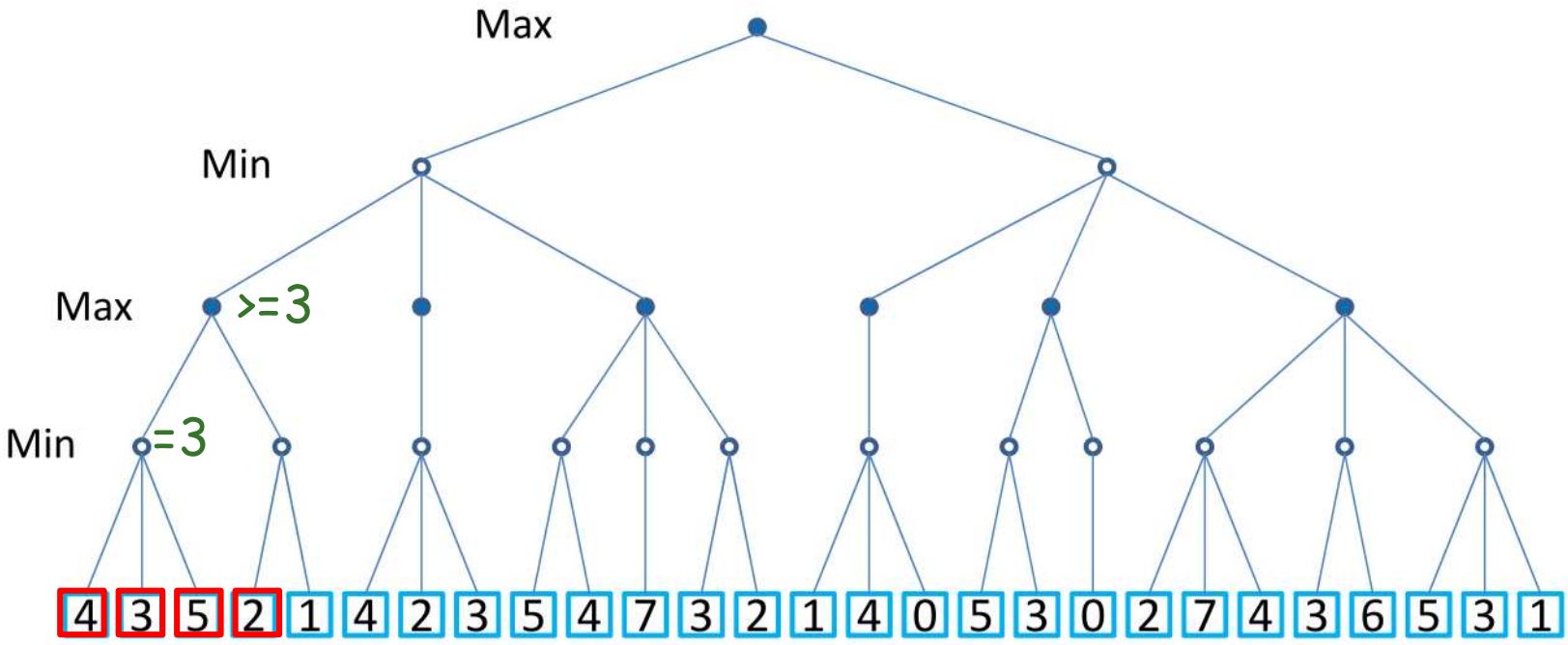
Alpha-Beta Pruning: Example 3



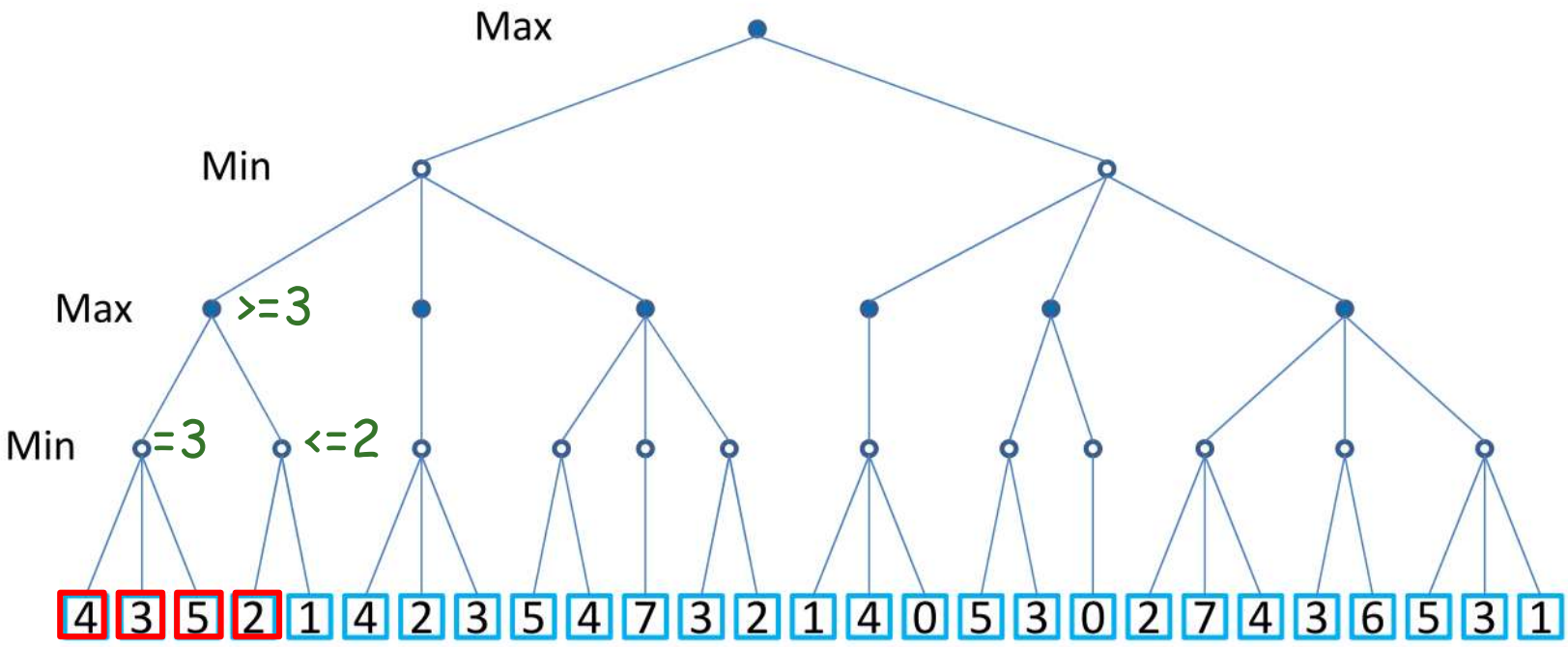
Alpha-Beta Pruning: Example 3



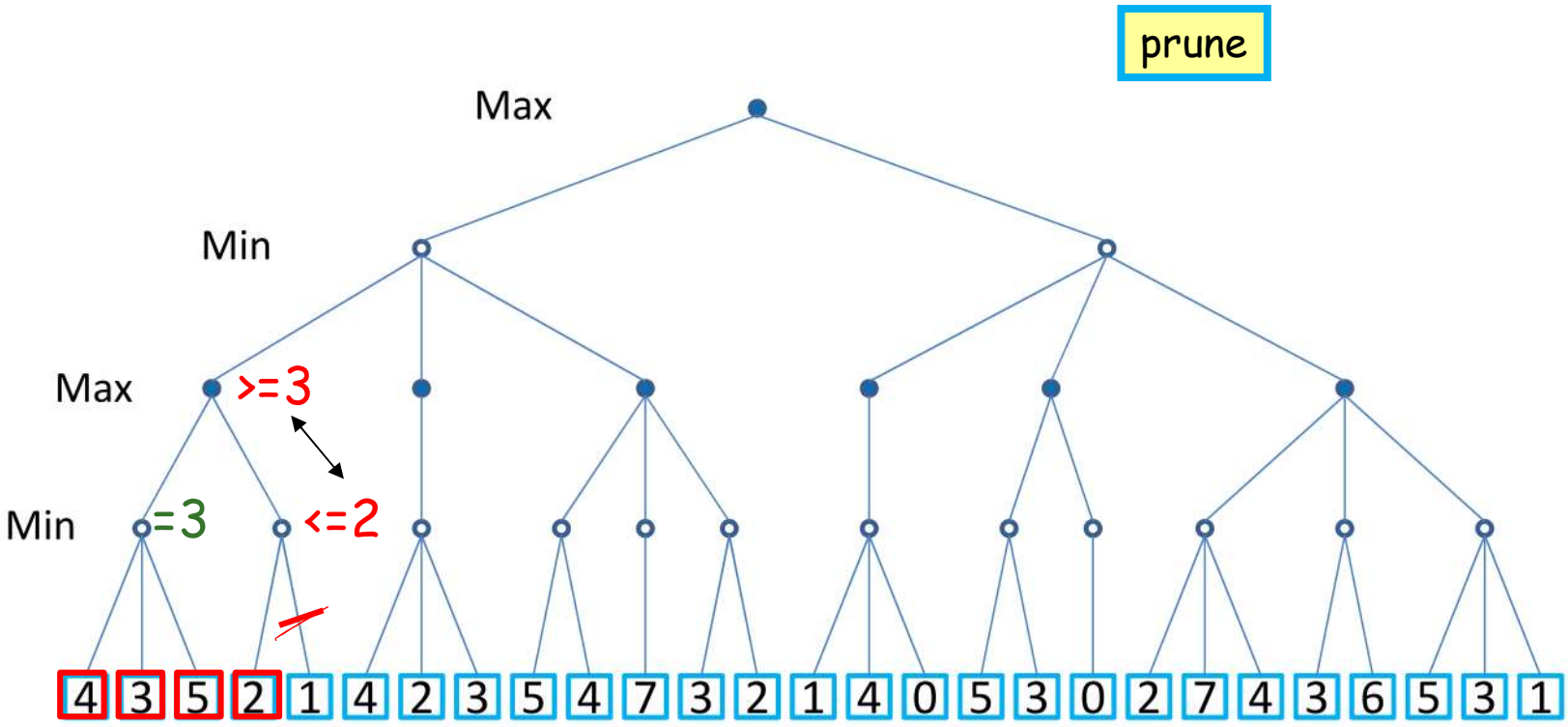
Alpha-Beta Pruning: Example 3



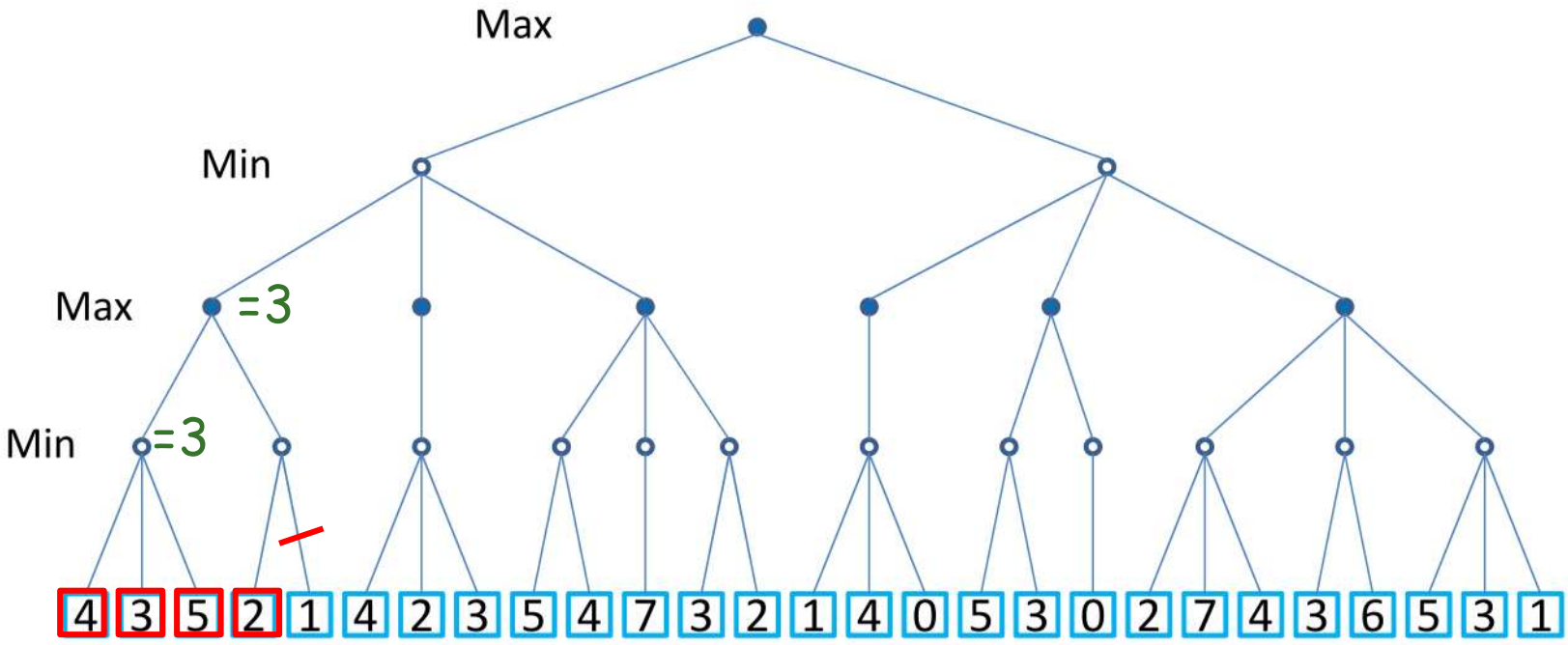
Alpha-Beta Pruning: Example 3



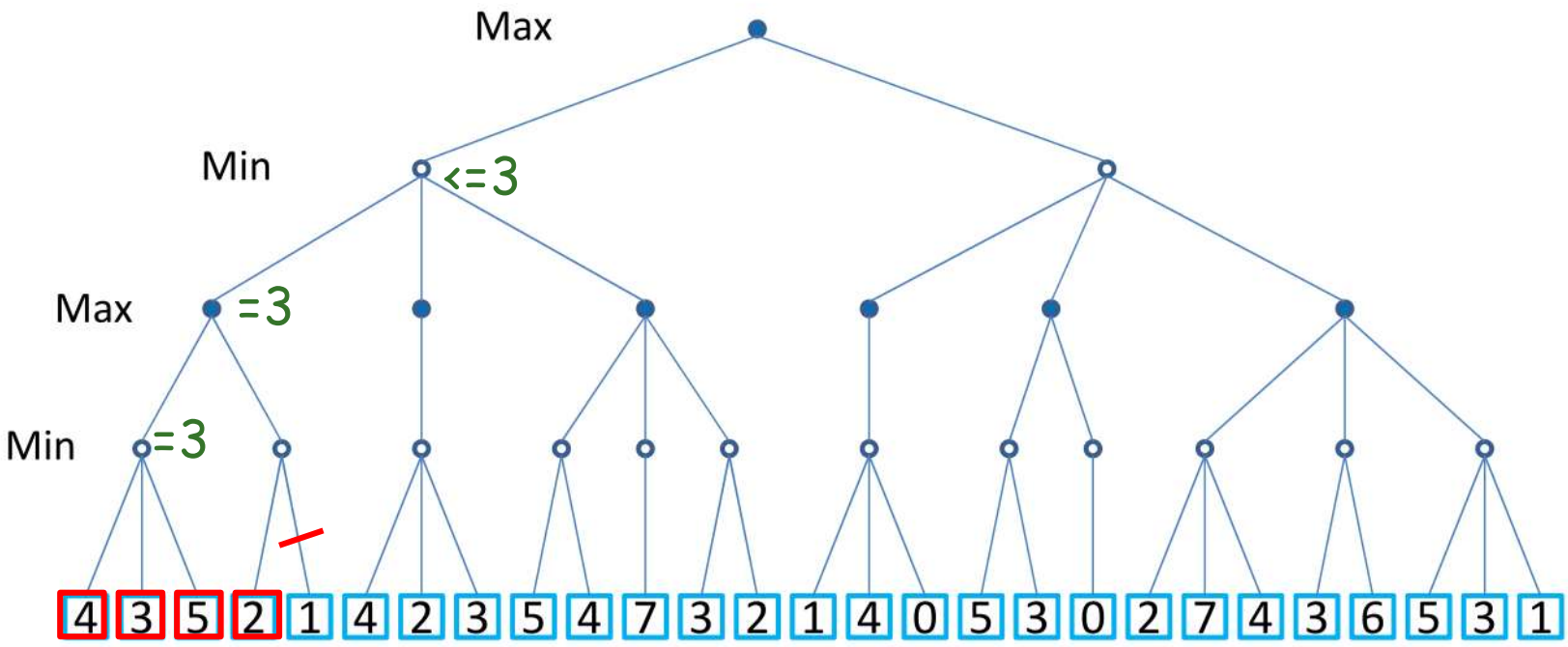
Alpha-Beta Pruning: Example 3



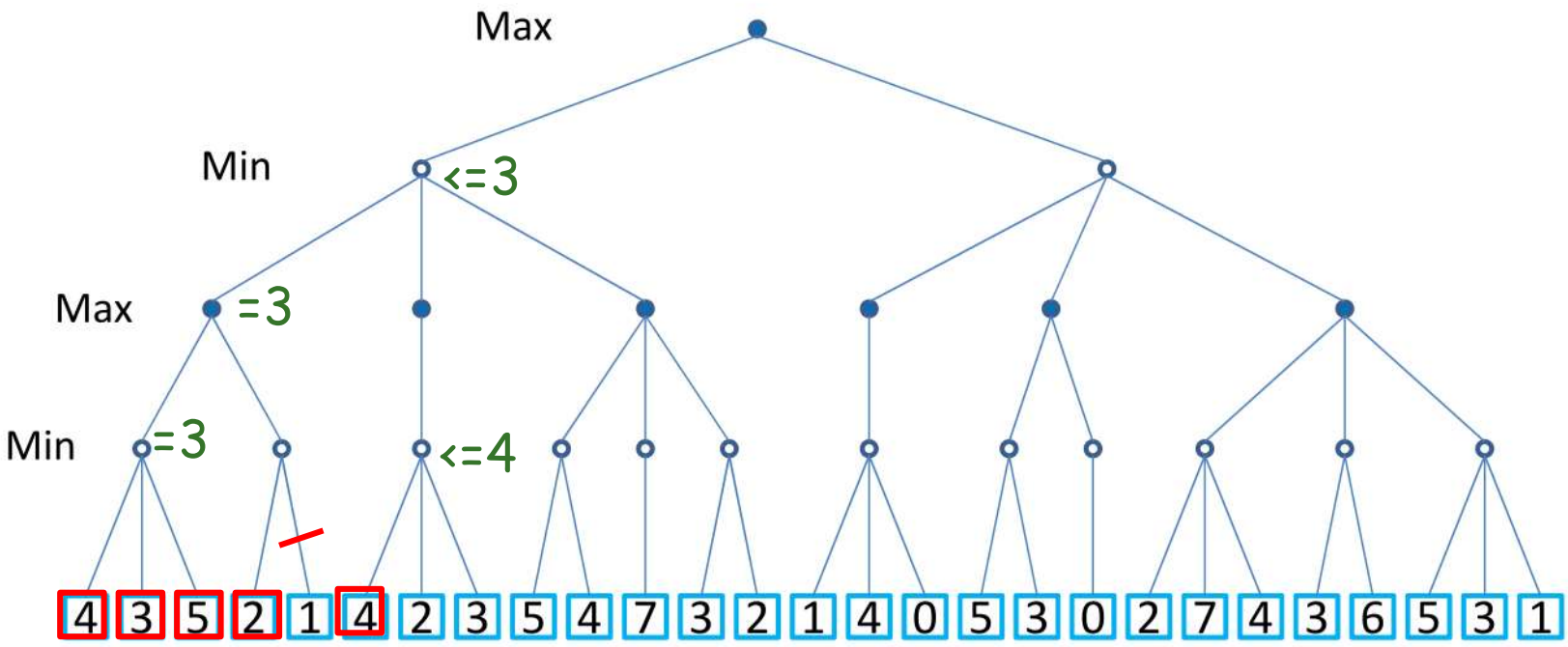
Alpha-Beta Pruning: Example 3



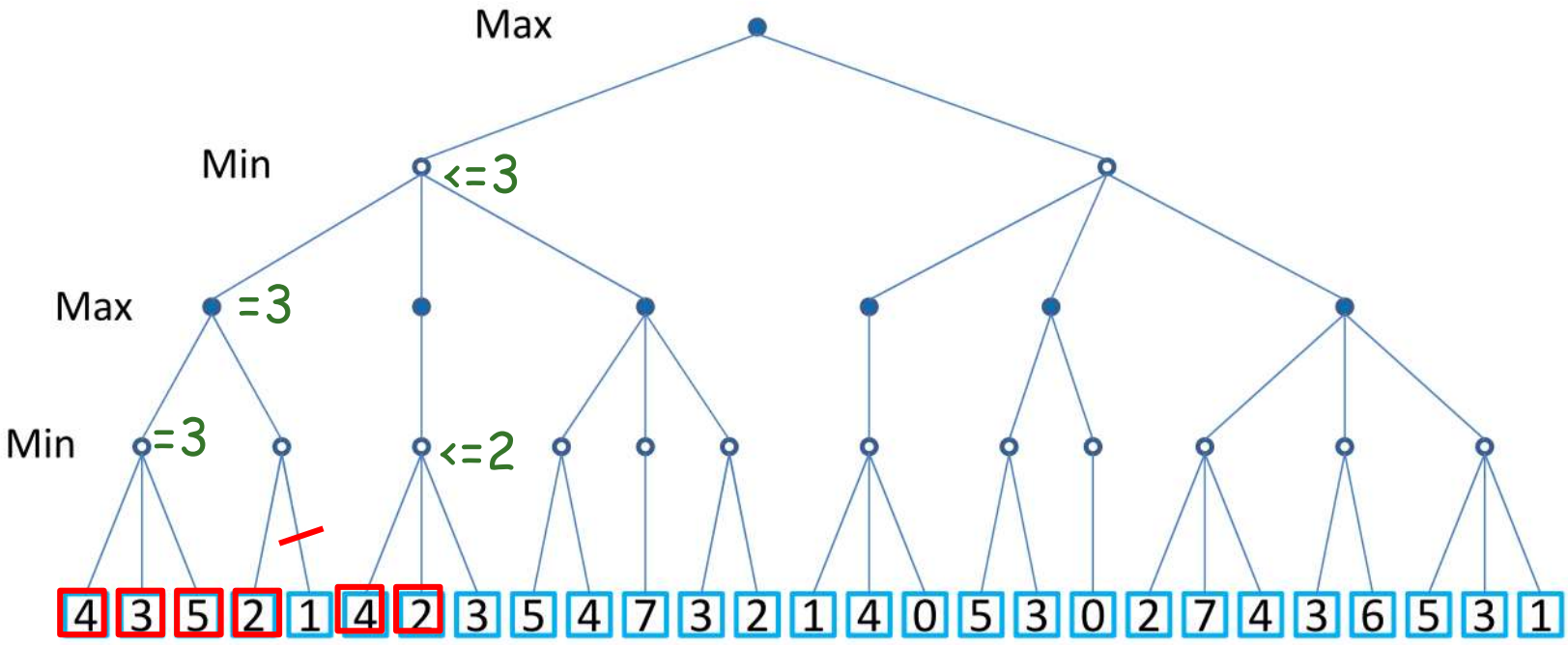
Alpha-Beta Pruning: Example 3



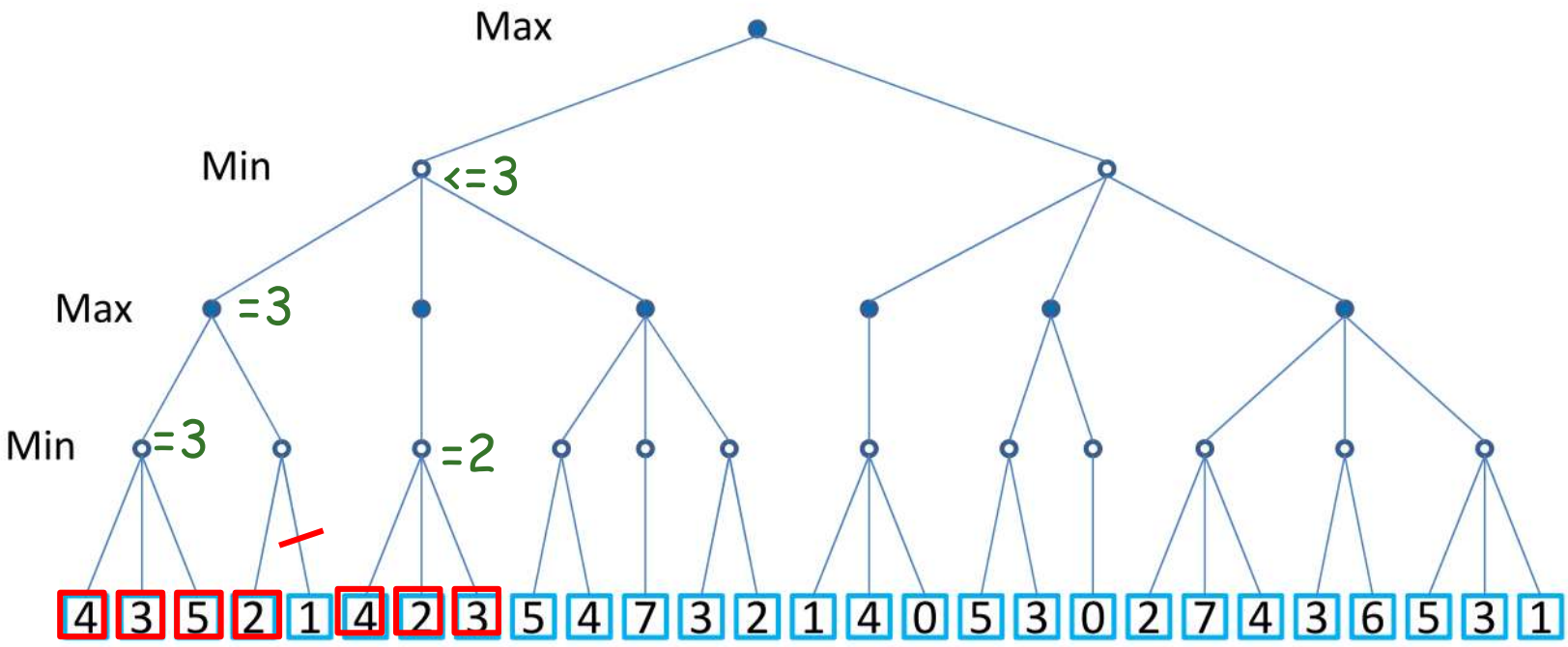
Alpha-Beta Pruning: Example 3



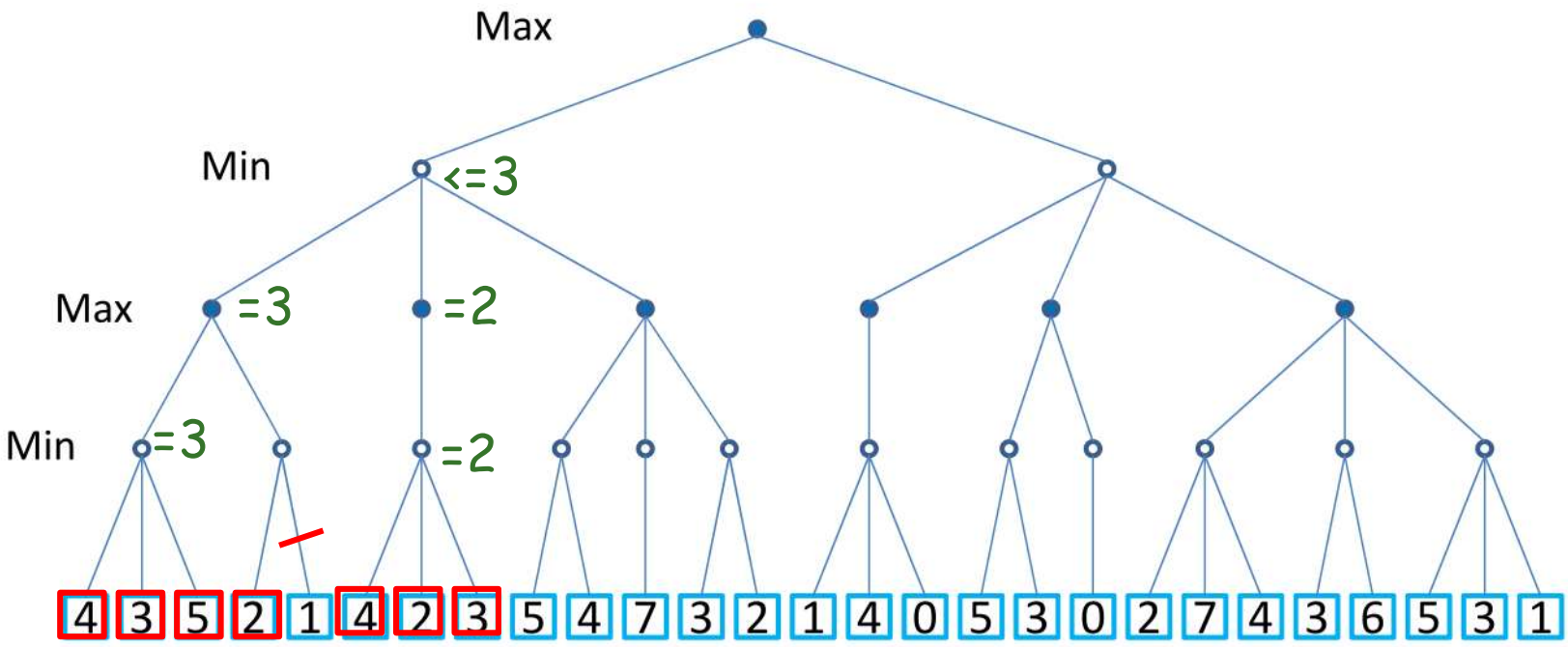
Alpha-Beta Pruning: Example 3



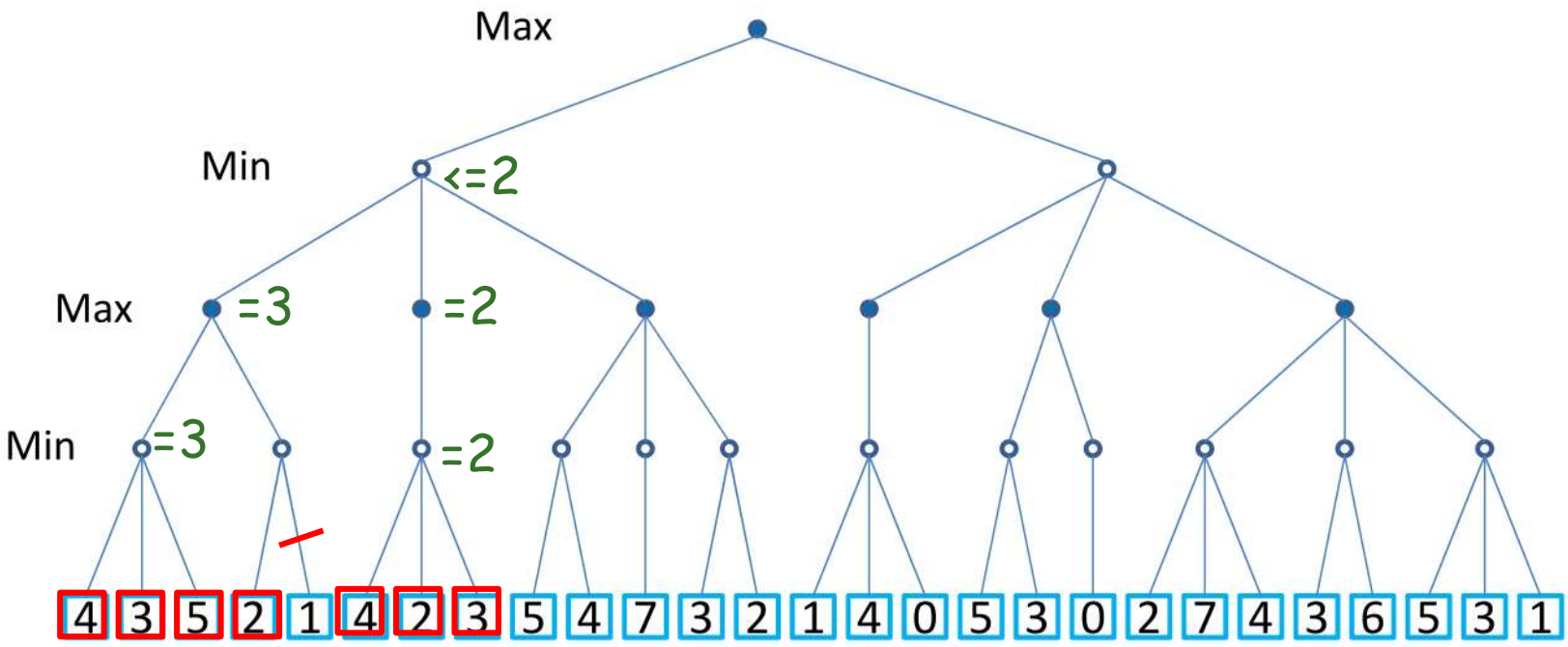
Alpha-Beta Pruning: Example 3



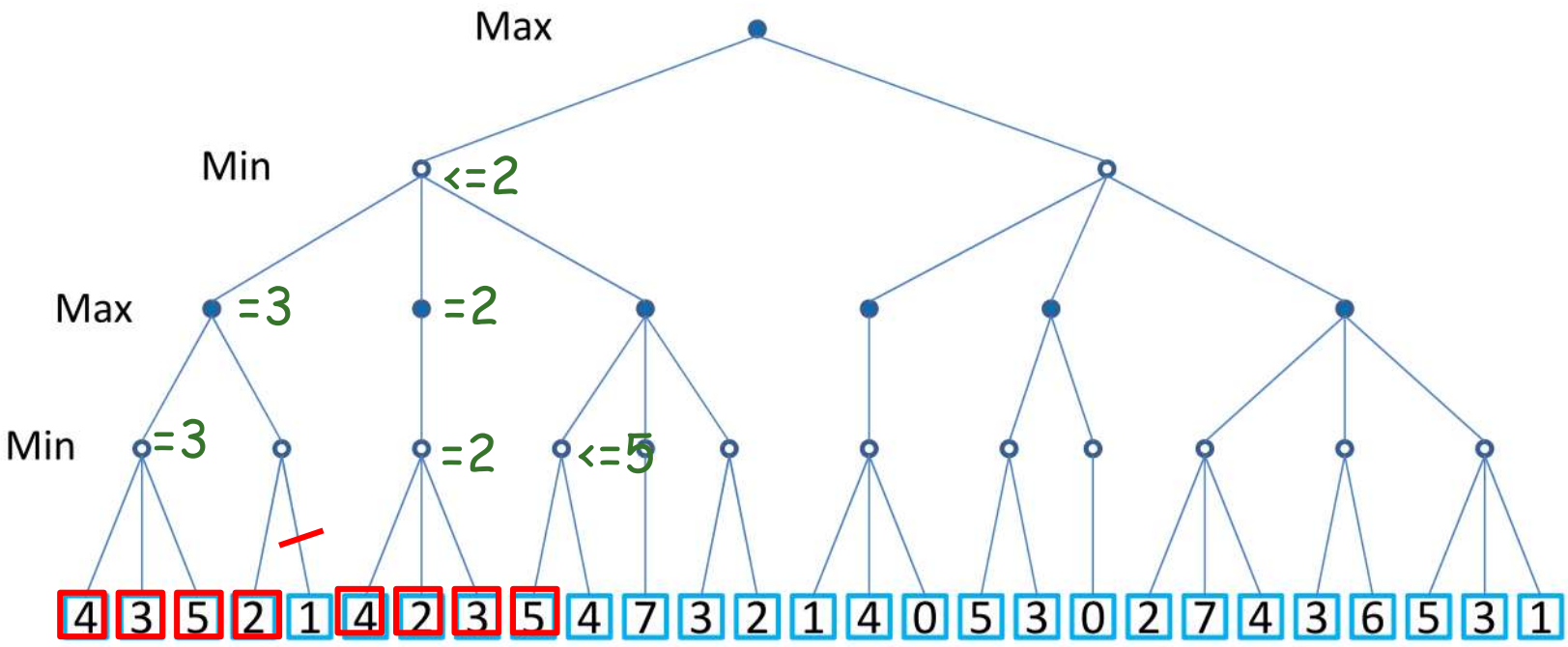
Alpha-Beta Pruning: Example 3



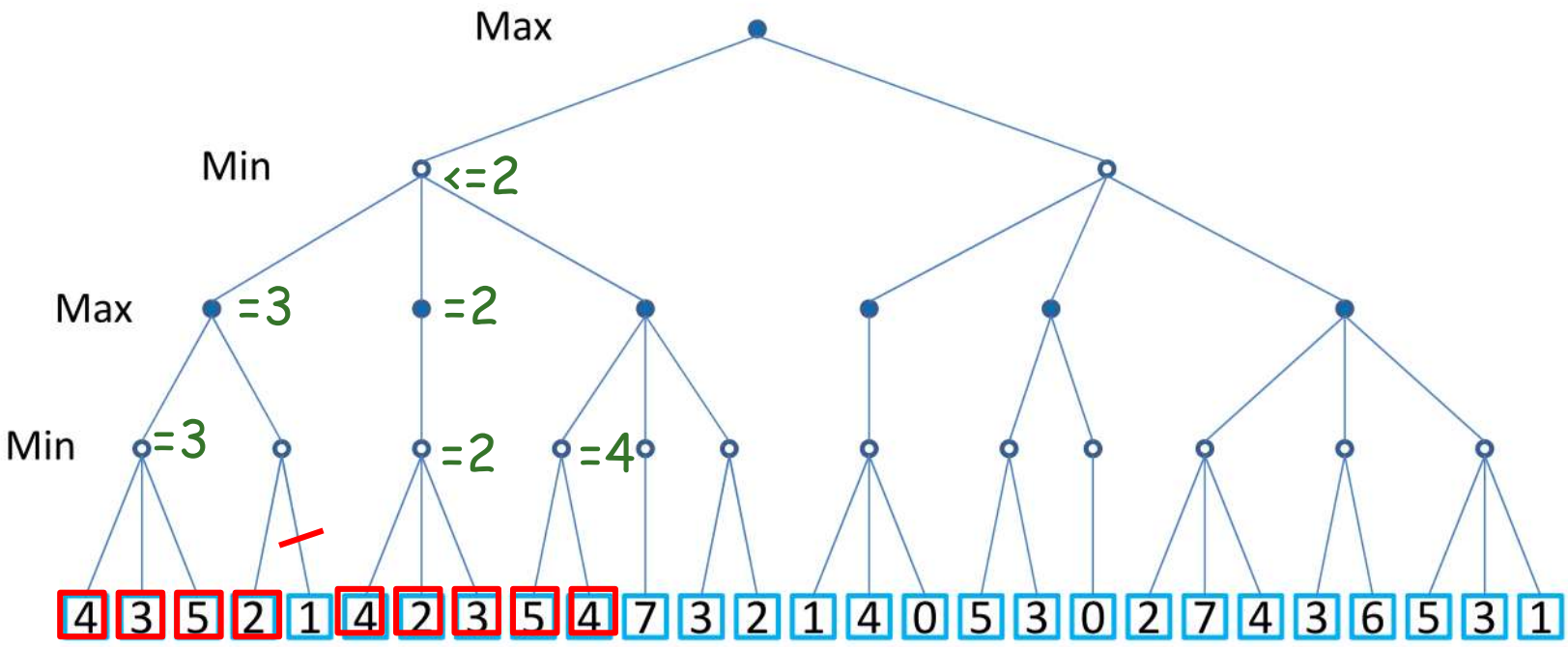
Alpha-Beta Pruning: Example 3



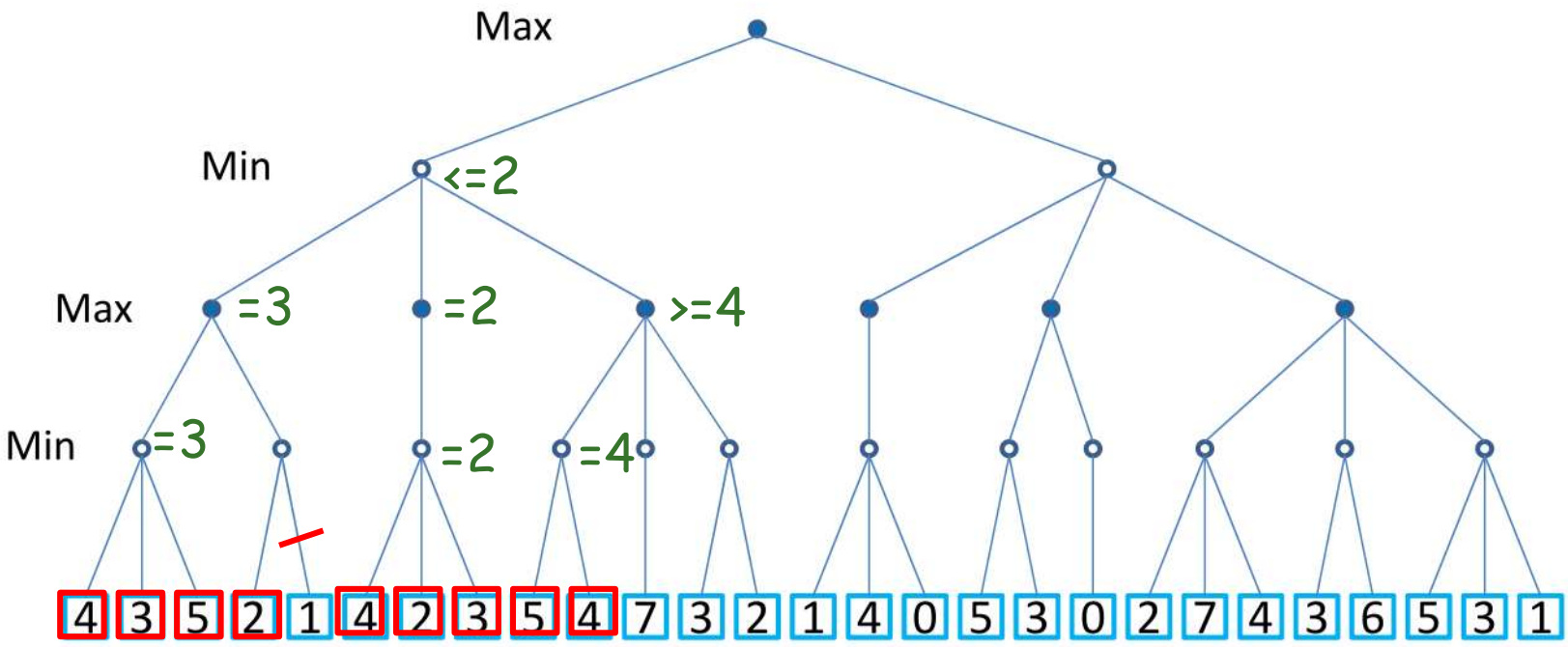
Alpha-Beta Pruning: Example 3



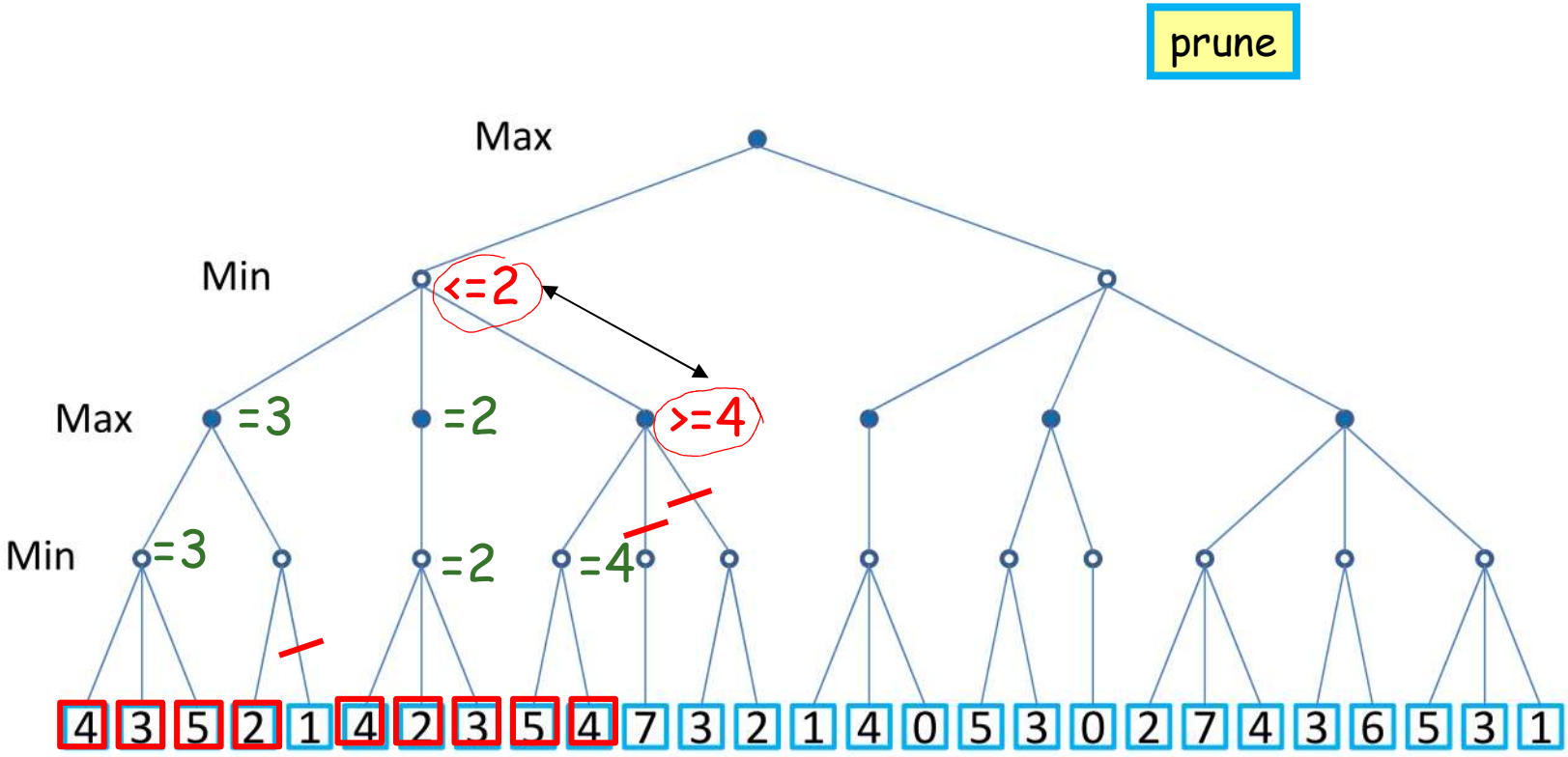
Alpha-Beta Pruning: Example 3



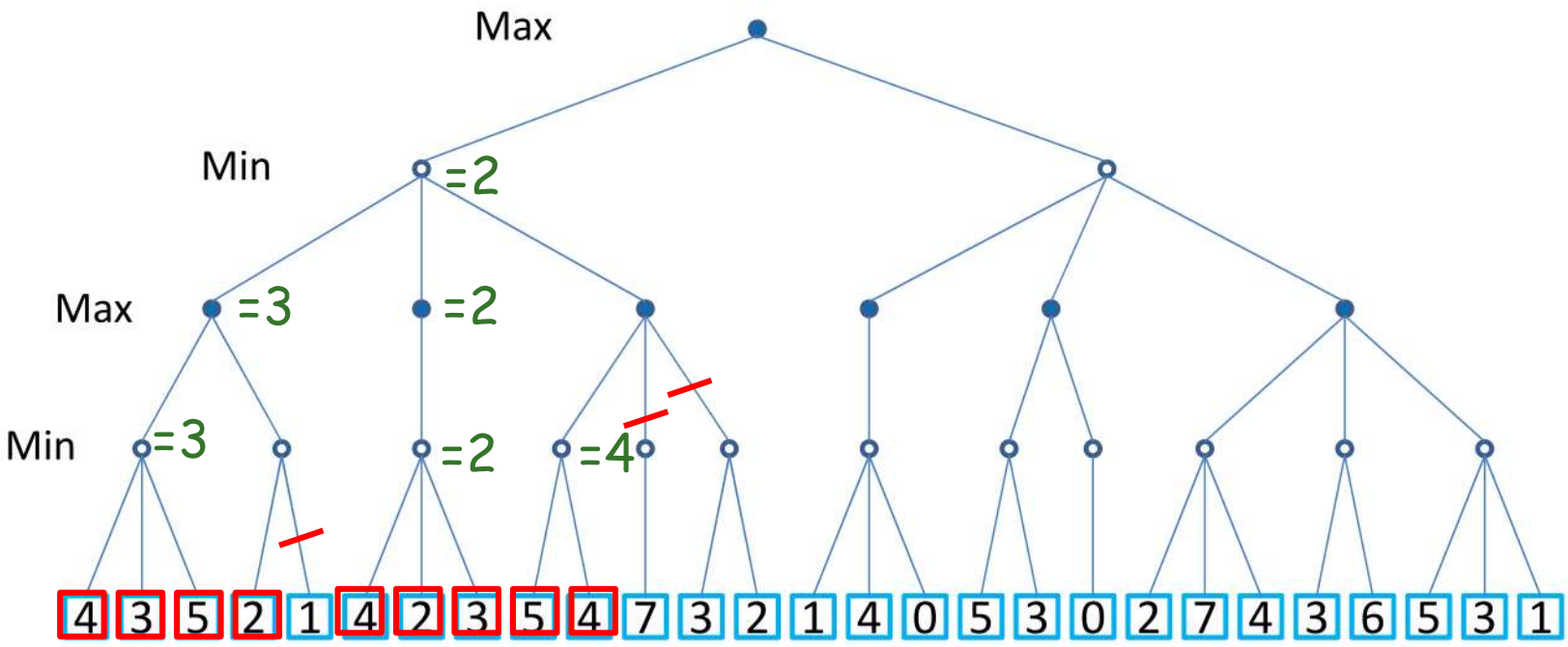
Alpha-Beta Pruning: Example 3



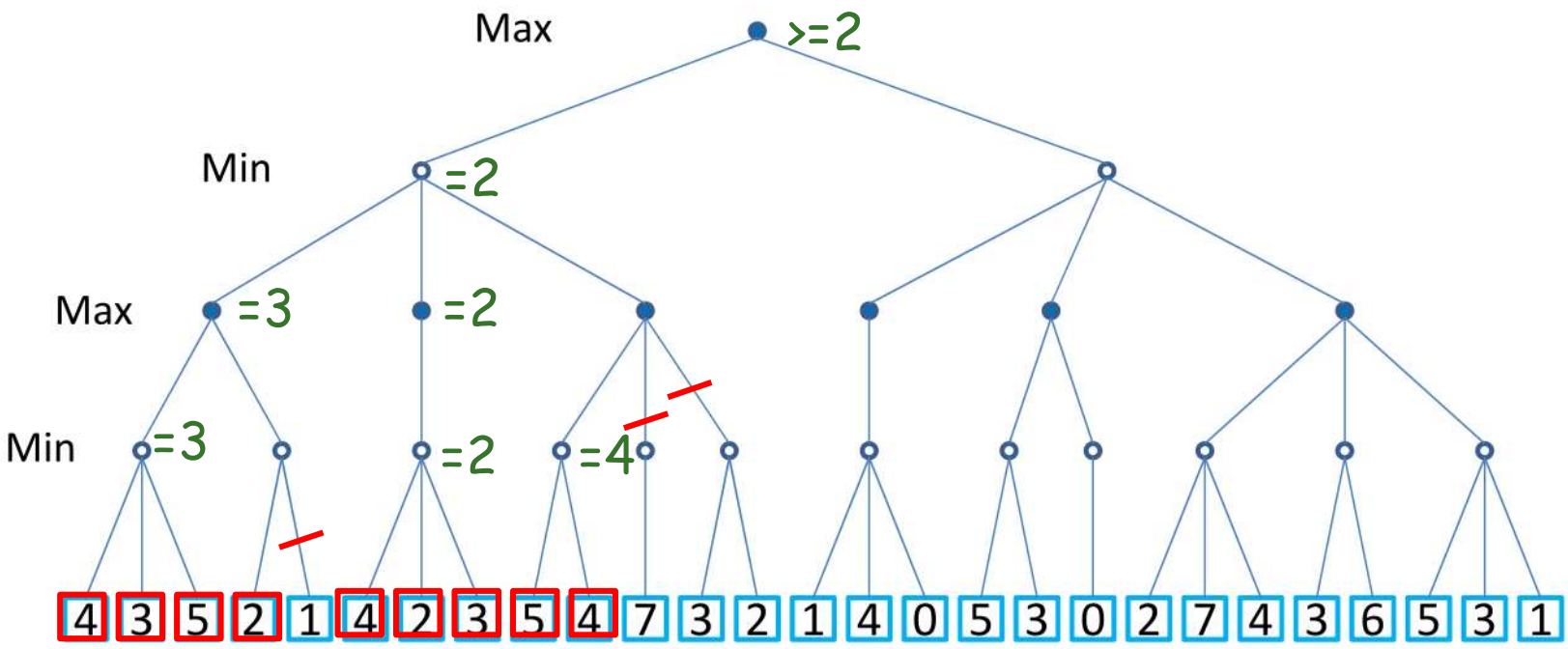
Alpha-Beta Pruning: Example 3



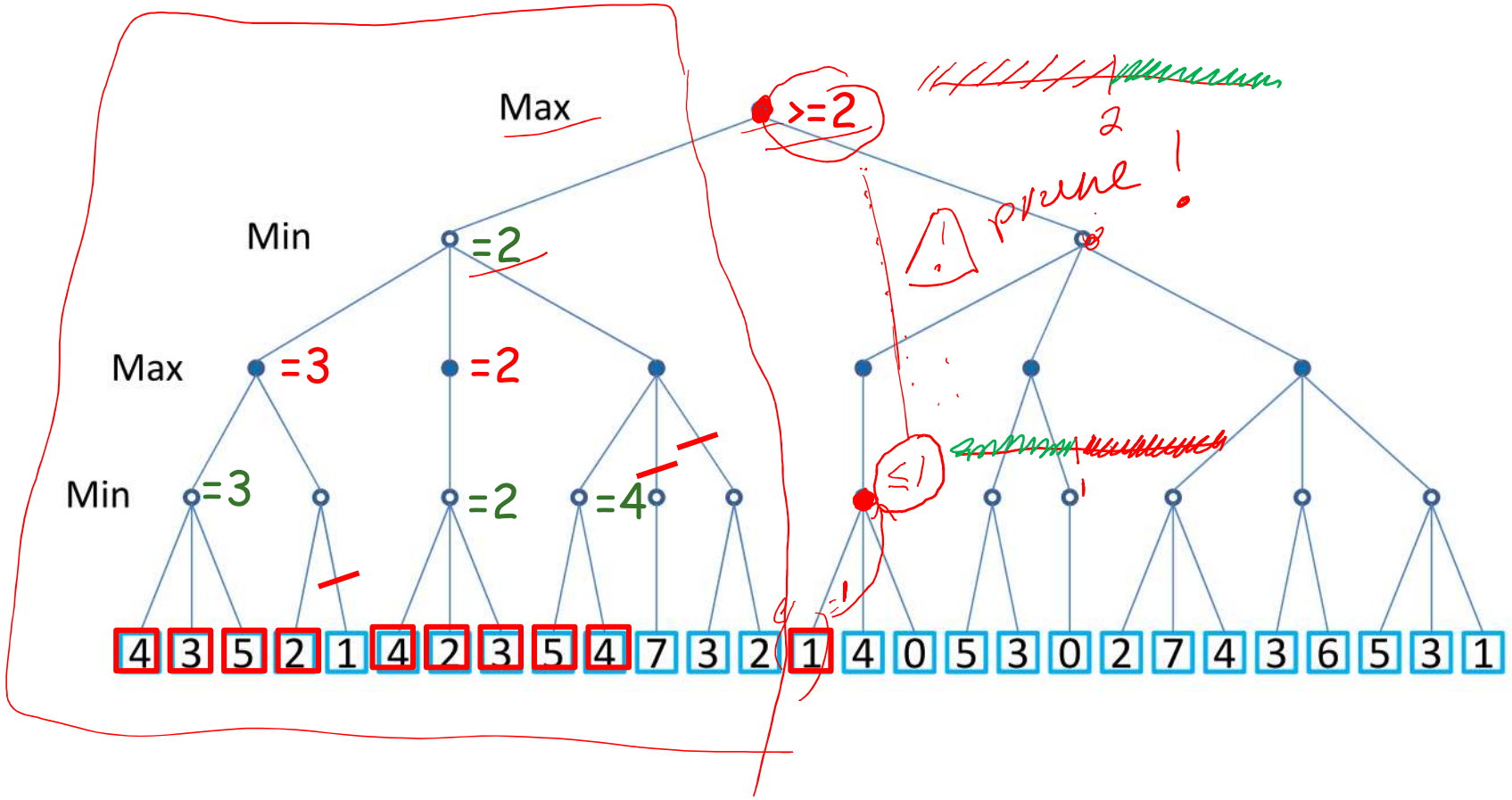
Alpha-Beta Pruning: Example 3



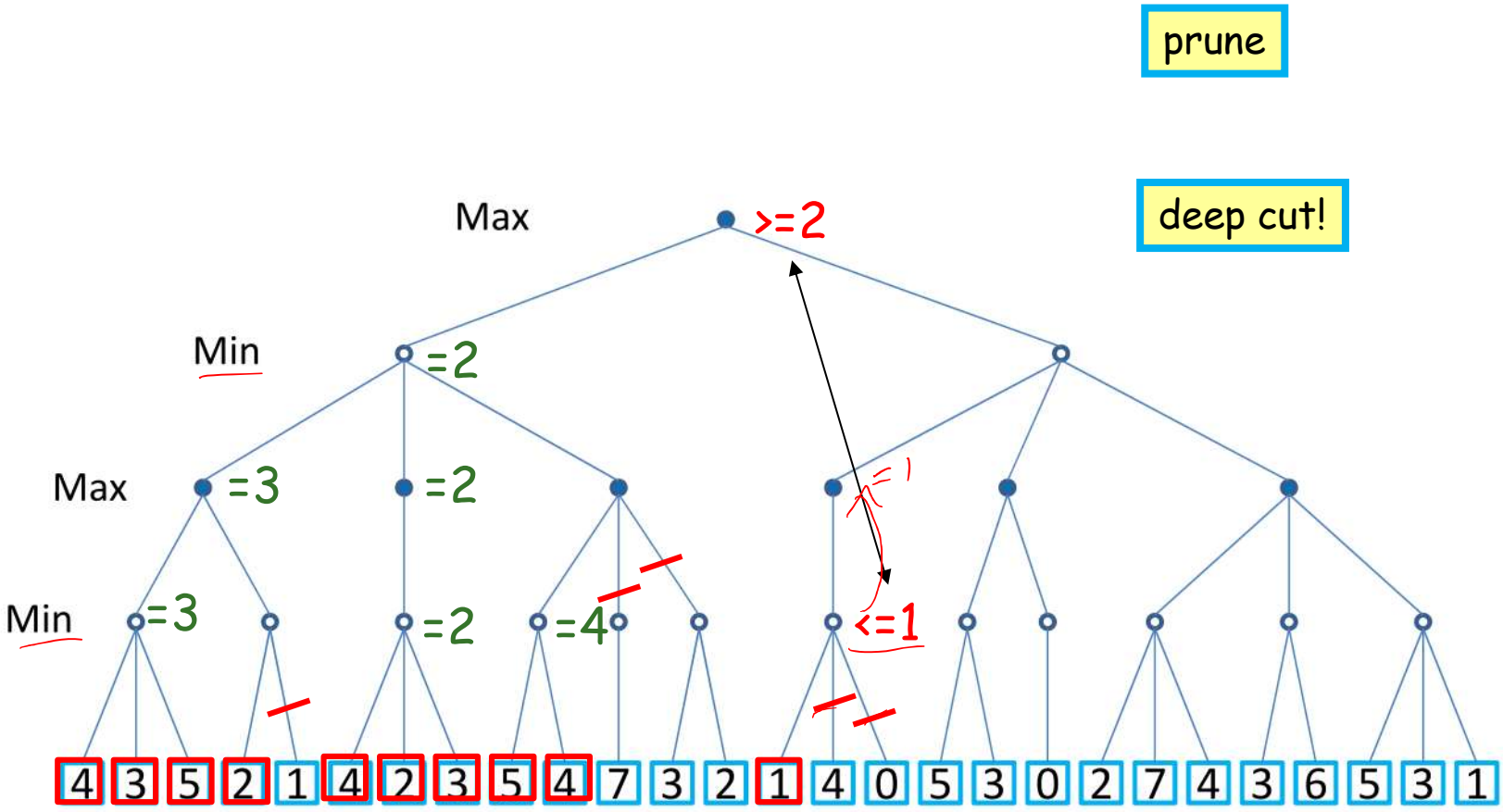
Alpha-Beta Pruning: Example 3



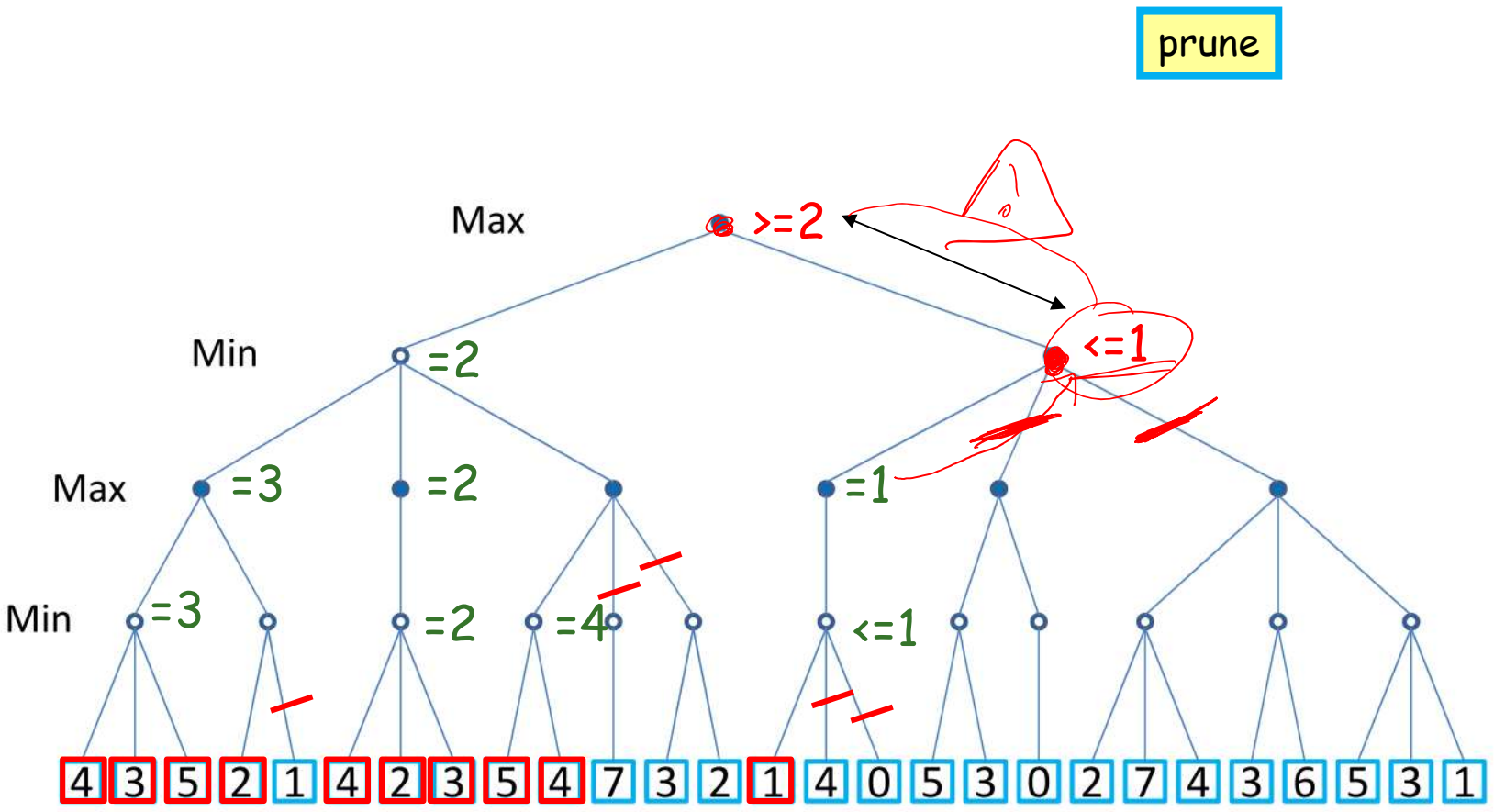
Alpha-Beta Pruning: Example 3



Alpha-Beta Pruning: Example 3



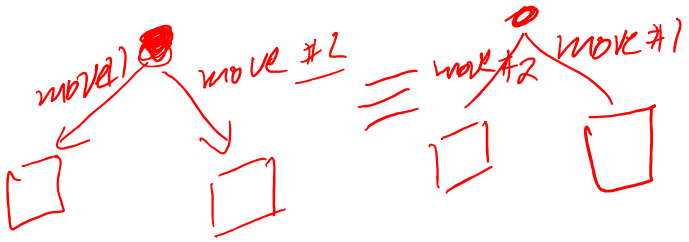
Alpha-Beta Pruning: Example 3



10 nodes explored out of 27

Efficiency of Alpha-Beta Pruning

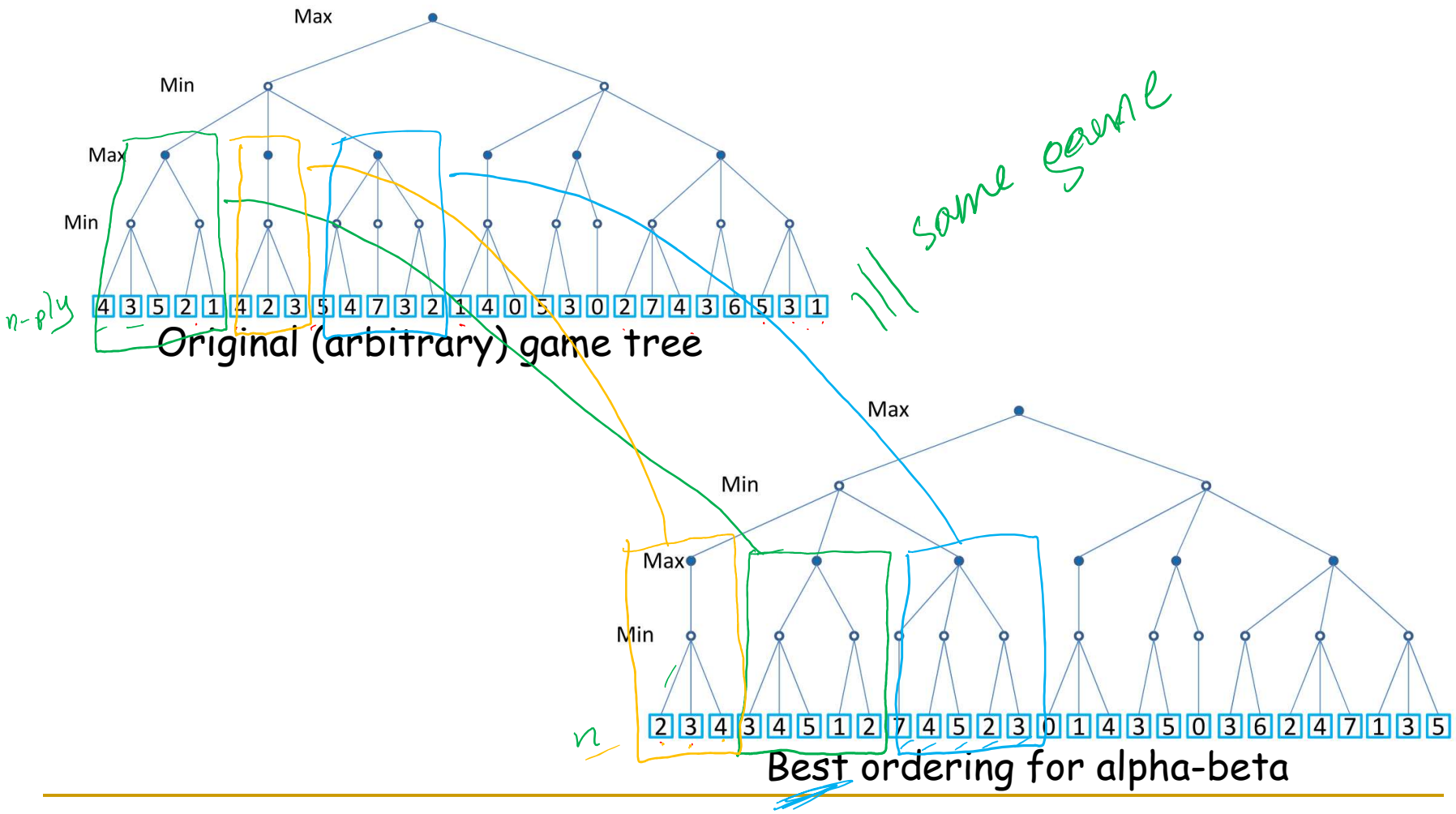
- Depends on the order the siblings
  - which is an arbitrary choice ;-(
- In worst case:
  - alpha-beta provides no pruning
  - plus extra overhead cost ;-(
- In best case:
  - branching factor is reduced to its square root



$36 \approx (n)$

$\Rightarrow 6 \approx (n)$

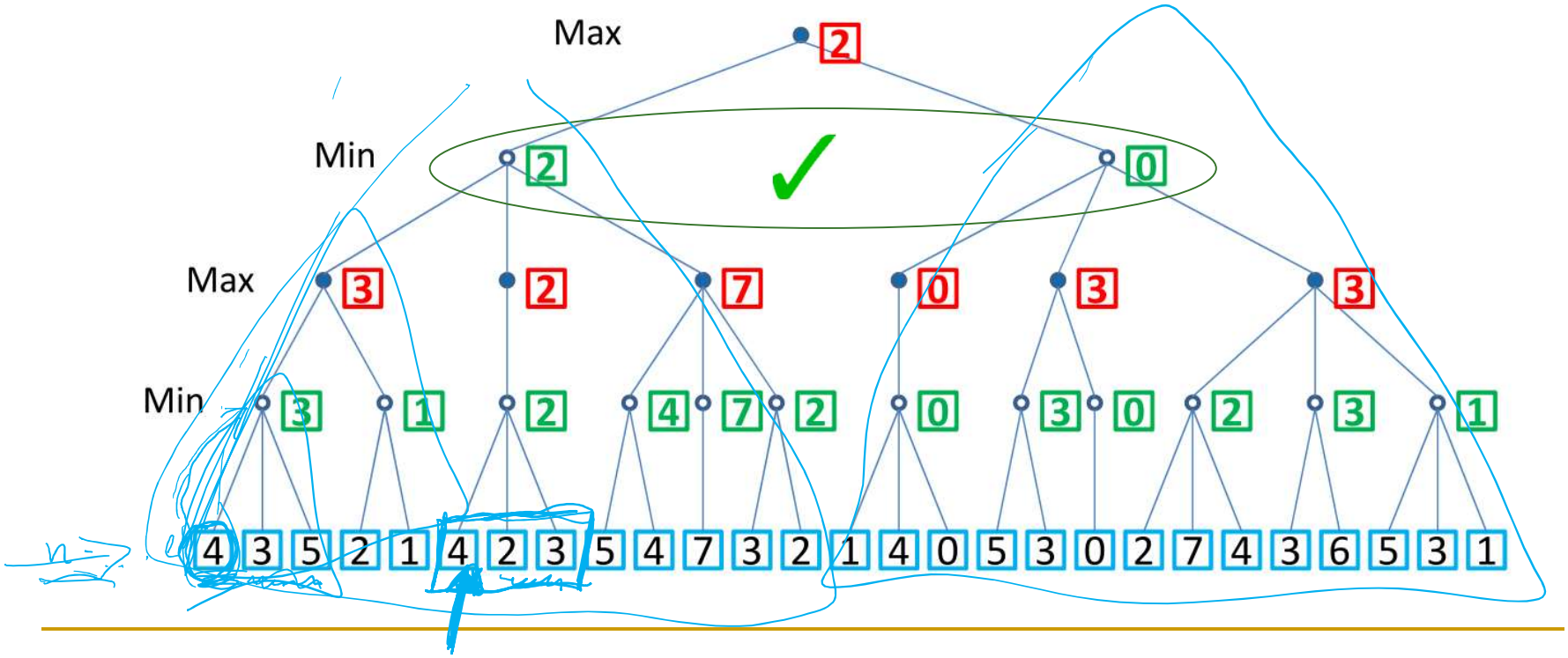
Alpha-Beta: Best ordering



Alpha-Beta: Best ordering

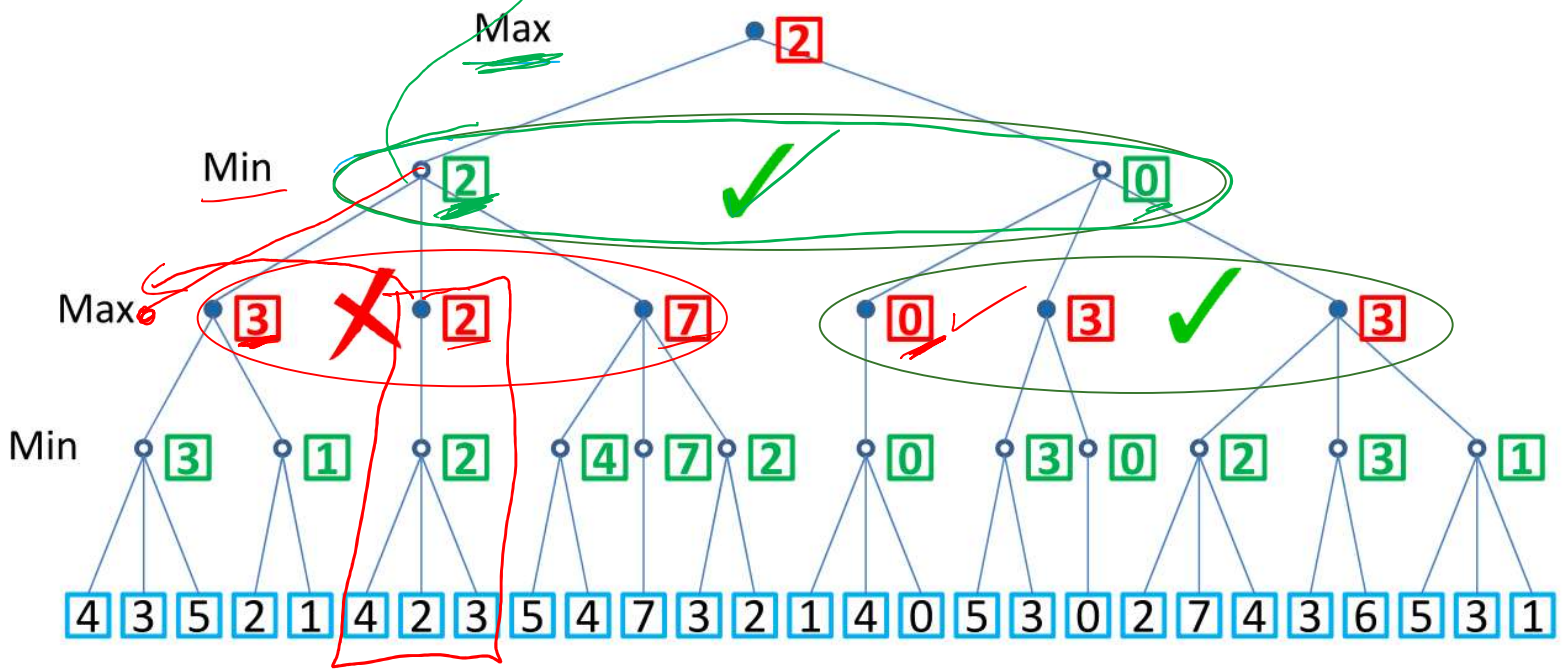
- best ordering:
  - strongest constraint placed first, ie:

1. children of MIN : smallest node first
2. children of MAX: largest node first



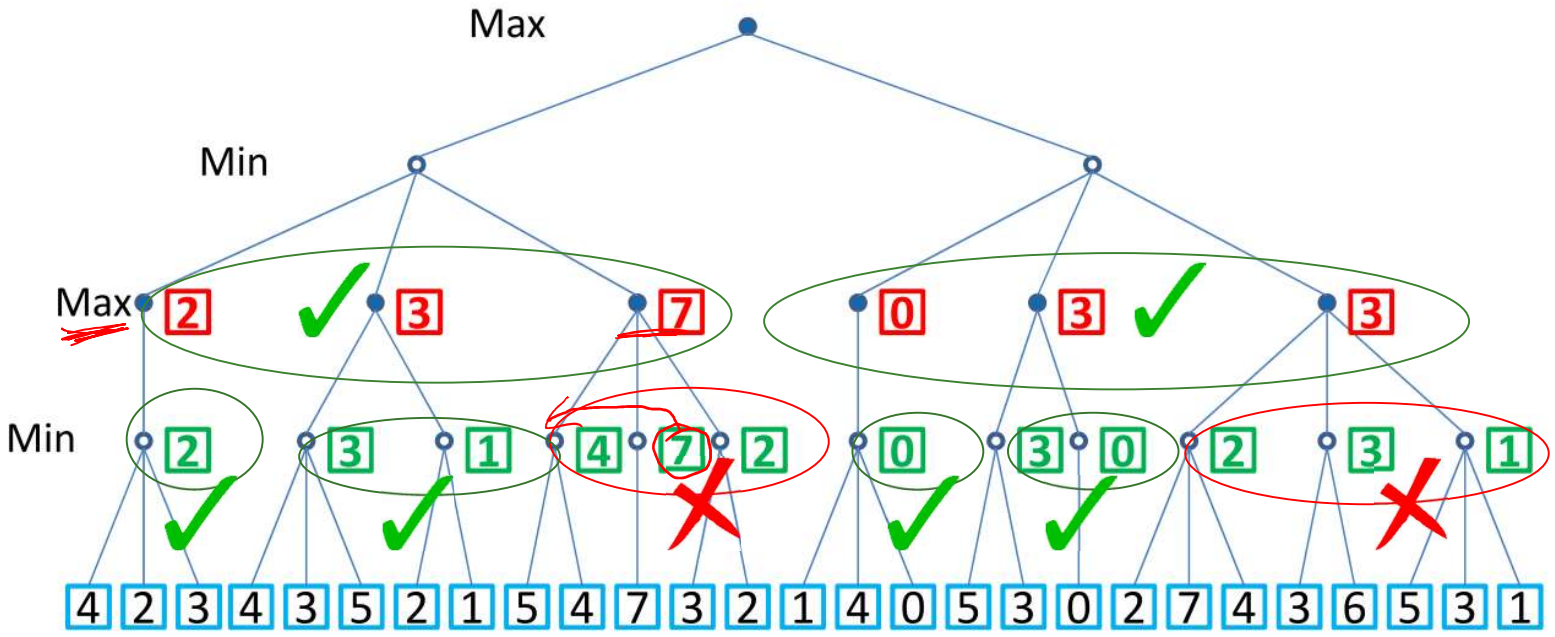
Alpha-Beta: Best ordering

- best ordering:
  1. children of MIN : smallest node first
  2. children of MAX: largest node first

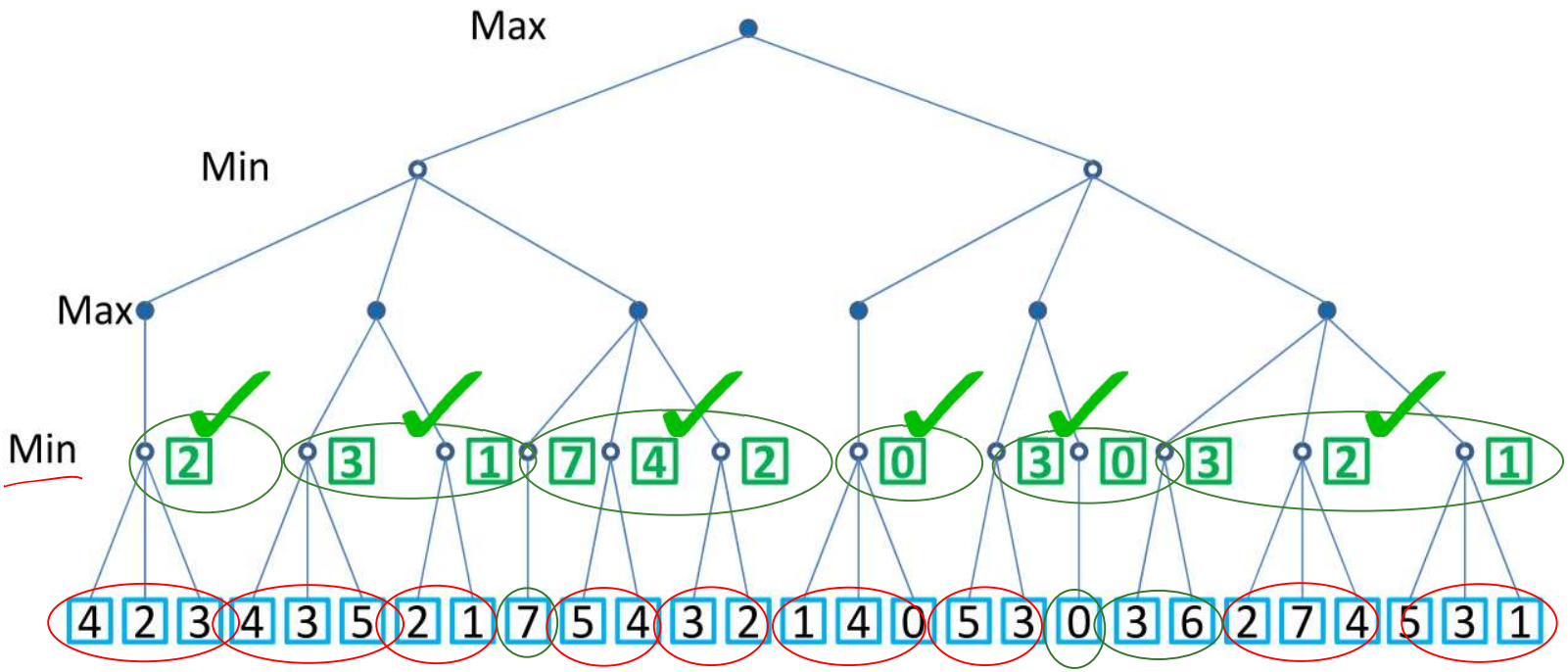


Alpha-Beta: Best ordering

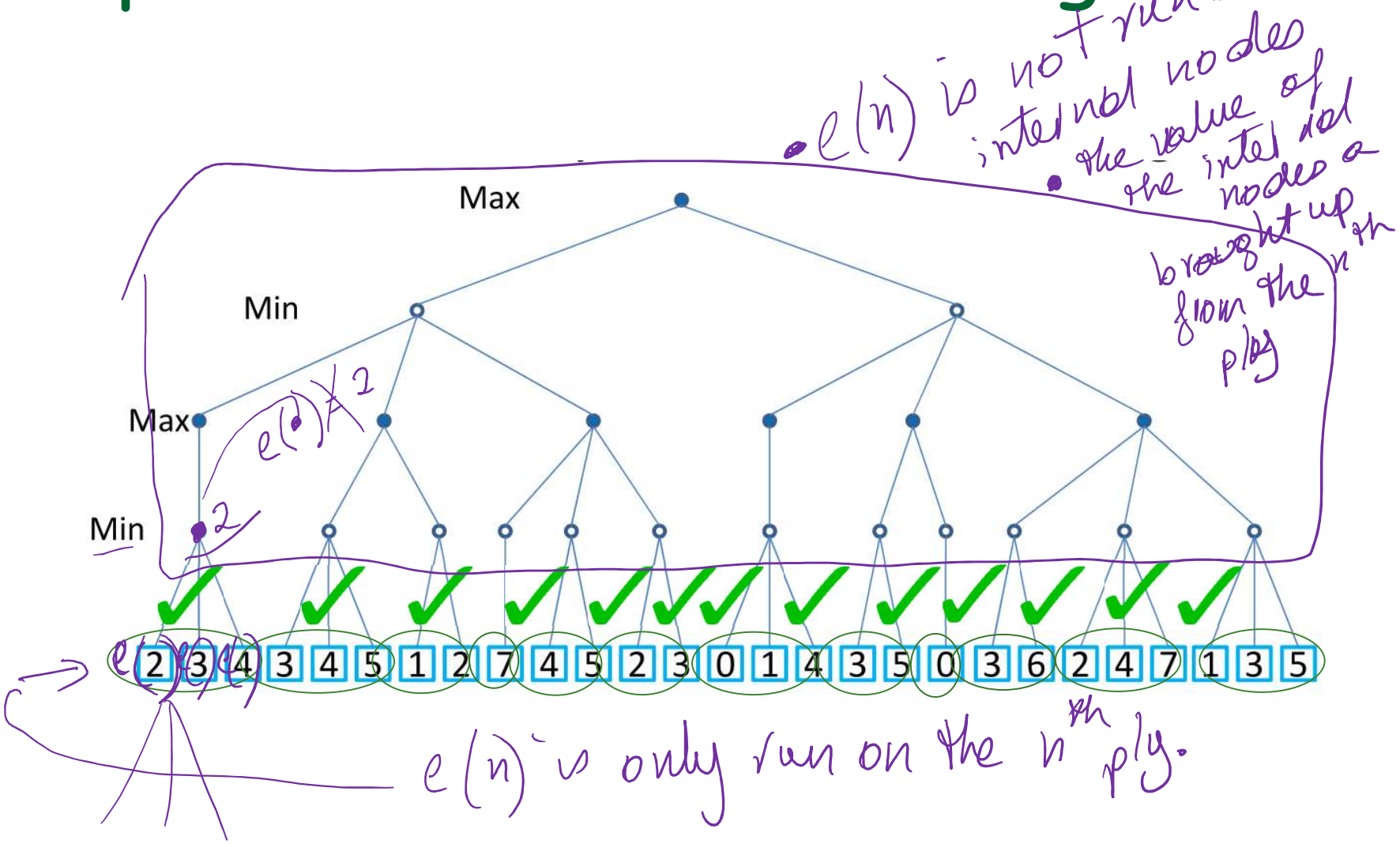
- best ordering:
  1. children of MIN : smallest node first
  2. children of MAX: largest node first



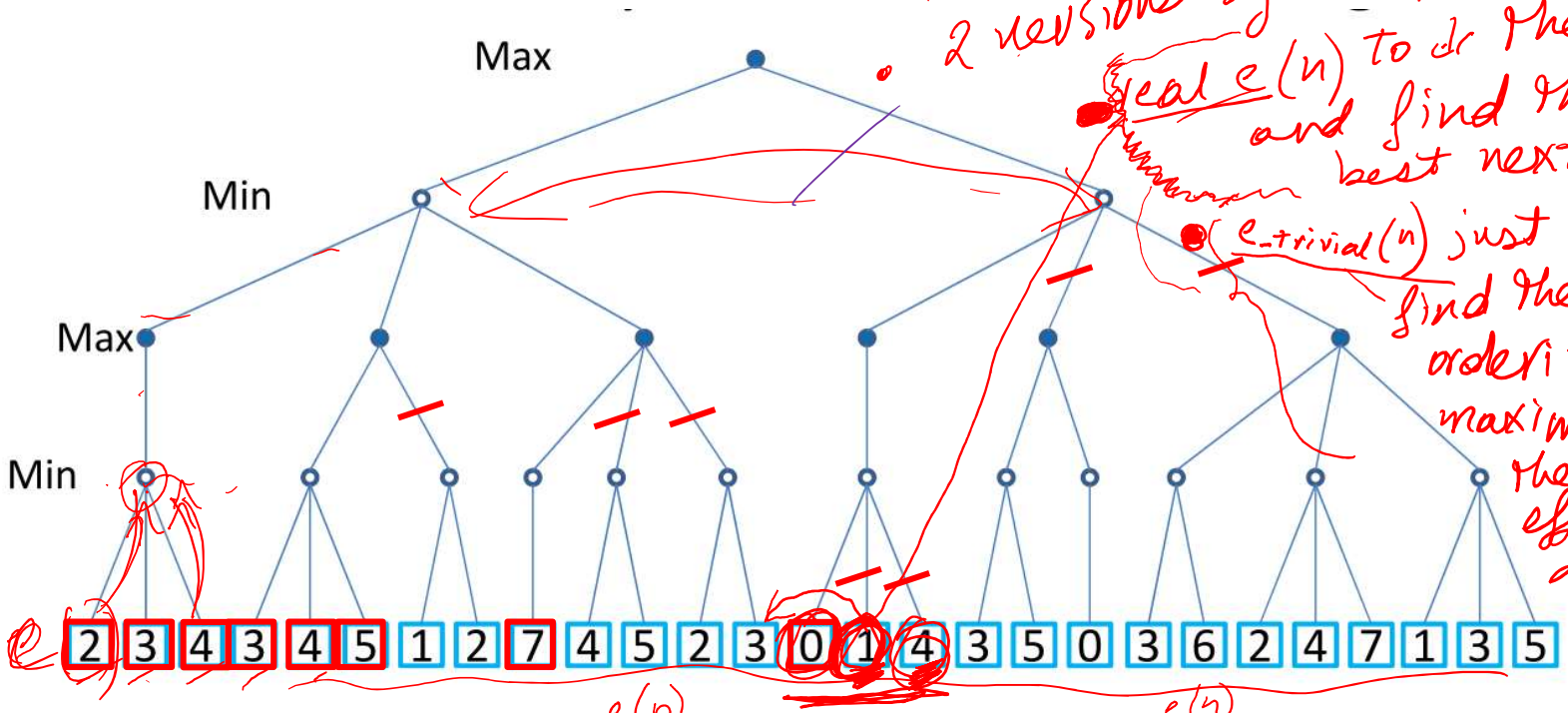
Alpha-Beta: Best ordering



Alpha-Beta: Best ordering



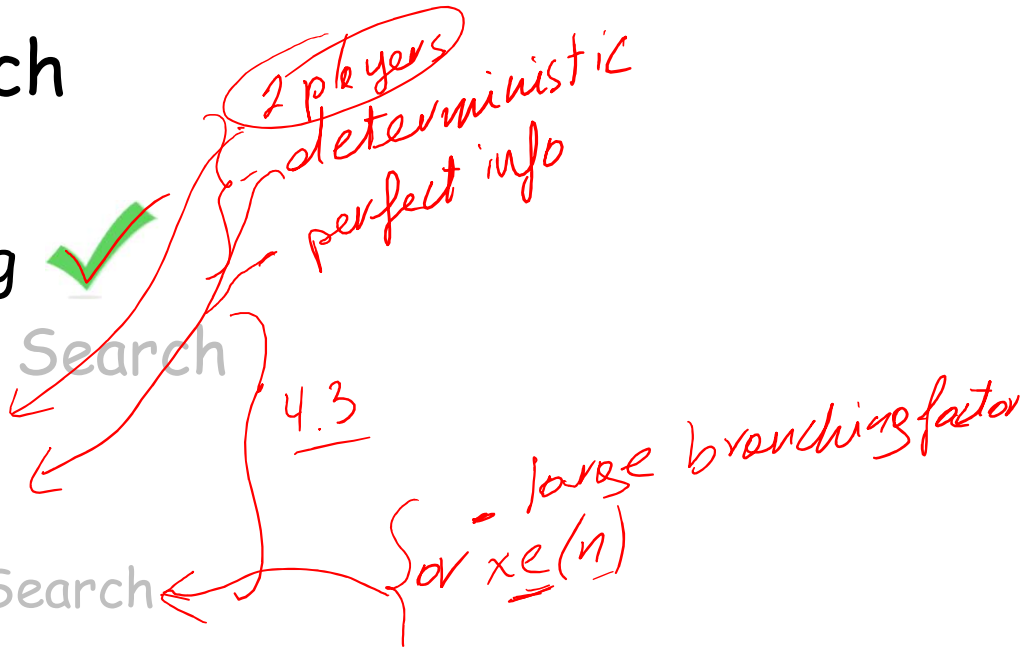
Alpha-Beta: Best ordering



8 nodes explored out of 27

Today

- Adversarial Search
  1. Minimax ✓
  2. Alpha-beta pruning ✓
  3. Other Adversarial Search
     1. Multiplayer Games
     2. Stochastic Games
     3. Monte Carlo Tree Search



Up Next

- Adversarial Search
  1. Minimax
  2. Alpha-beta pruning
  3. Other Adversarial Search
     1. Multiplayer Games
     2. Stochastic Games
     3. Monte Carlo Tree Search