复旦大学大数据学院
School of Data Science, Fudan University

Lab 1: Alpha-Beta Pruning & CSPs

魏忠钰

April 5th 2017

Python Setup

- **Python 2.7 Download:**
  - https://www.python.org/downloads/release/python-2712/
- **Python 2.7 Tutorial:**
  - https://docs.python.org/2/tutorial/
- **Python IDE:**
  - **Pycharm**
    https://www.jetbrains.com/pycharm/download/
  - **Spyder Python**
    http://www.discoversdk.com/products/spyder-python

Outline

- Alpha-beta pruning
  - Submit in class via OJ
- Constraint Satisfaction Problems
  - Take home as an assignment (Homework 2)

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

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

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

Alpha-Beta Implementation


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

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

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

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Demo: Alpha-beta pruning

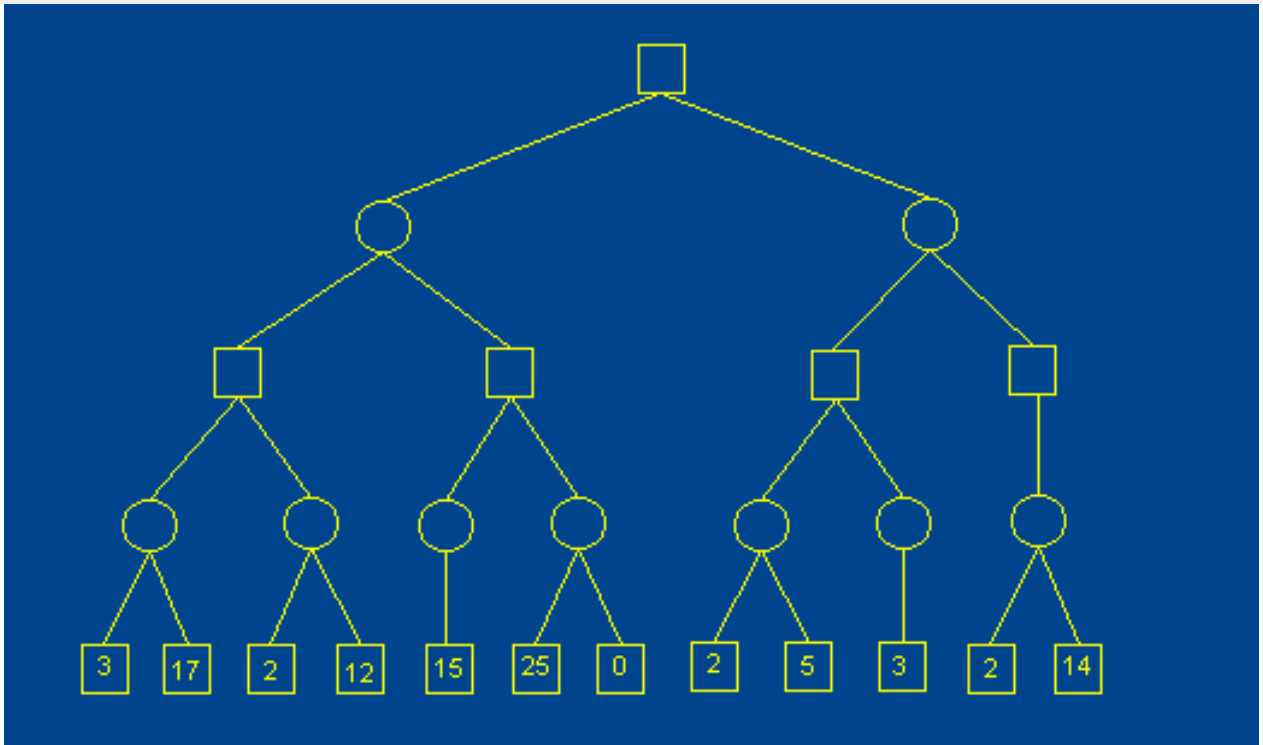
http://inst.eecs.berkeley.edu/~cs61b/fa14/ta-materials/apps/ab_tree_practice/

Practice for Alpha-beta Pruning

Problem 1174: http://10.88.12.45/JudgeOnline/problem.php?id=1174

Input Example: **Square represents MAX node while circle stands for MIN node.** For each test case, it contains two lines. The first line consists of two integers, the role of the root node (1 for MAX node and 0 for MIN node) and the depth of the tree. The second line is a nested list which stands for the game tree. The sample input shows the tree of Fig1

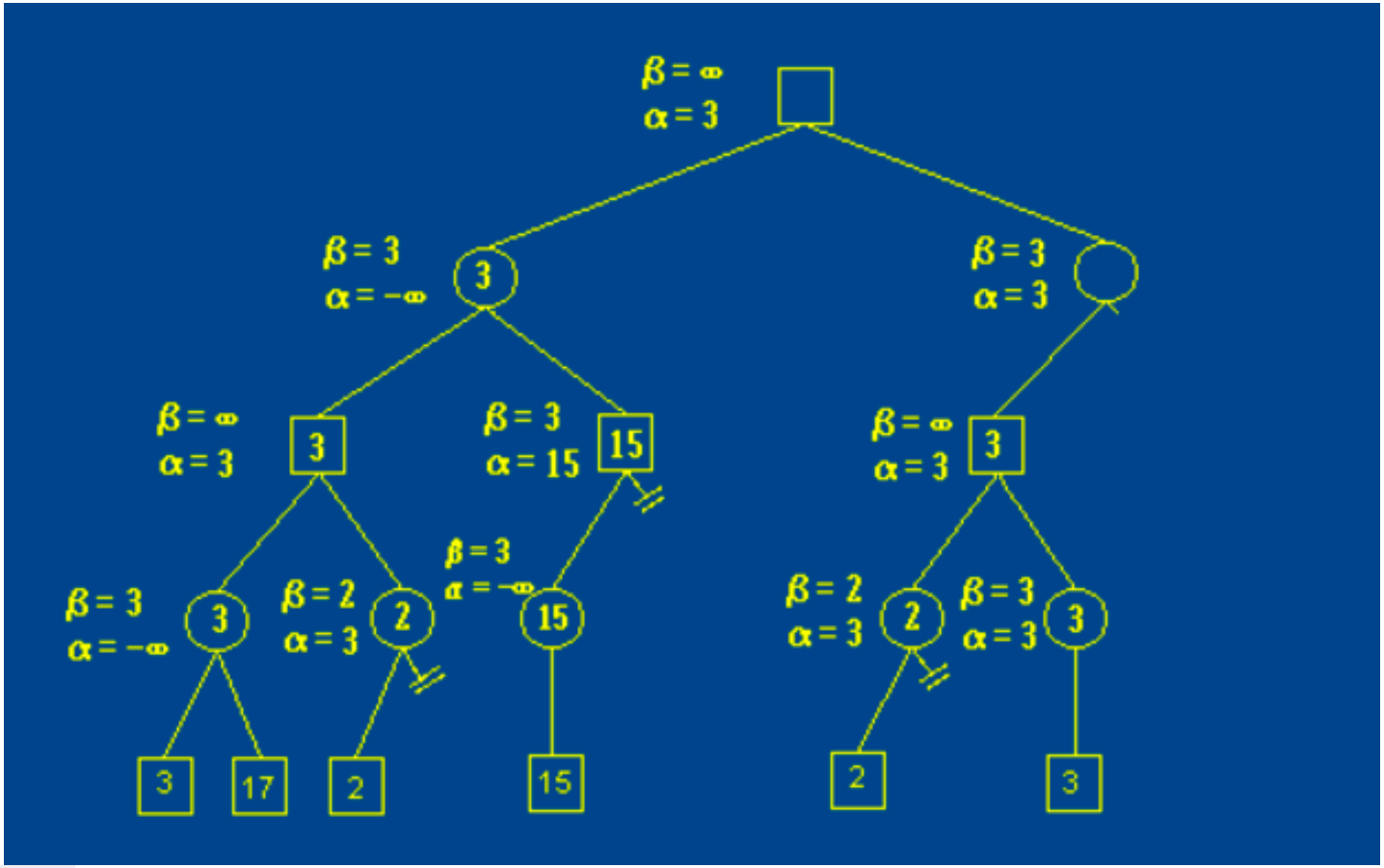
```
1 5
[[[[3,17],[2,12]],[[15],[25,0]]],[[[2,5],[3]],[[2,14]]]]
```



Practice for Alpha-beta Pruning

Problem 1174: http://10.88.12.45/JudgeOnline/problem.php?id=1174

For each test case, the output should include two lines. The first line contains the result for minimax search. The second line should consist of **pruned nodes** in order.




```
3
12 25 0 5 2 14
```

ab-pruning Template

- Hint()
  - Eval (): Read the string and store it as a python type
  - i.e. tree = eval(raw_input().strip())
  - Mark tree nodes which have been visited
- **def** value(node, alpha, beta)
  - Choose which function to call
- **def** maxValue(node, alpha, beta)
- **def** minValue(node, alpha, beta)

HW-2 for CSP

- Backtracking is a basic solution for CSP problem #problem a
- You can improve the performance of CSP by 3 ways:
- Ordering:
  - Which variable should be assigned next? (MRV) #problem b
  - In what order should its values be tried? (LCV)
- Filtering: Can we detect inevitable failure early?
  - Arc-consistency checking (AC -3) #problem c
- Structure: Can we exploit the problem structure? Not this time
  - Tree structure
  - Cutting set conditionning

HW-2 for CSP

- You need to submit your own version of code.
- You are encouraged to discuss with your group members. It might take some time to get familiar with all the supportive codes.
- **Homework 2 is due on April 19th, Wednesday, 11:59pm, 2017.**
- **Download description here.**
- **Get code here.**

Util.py


```
class CSP{

    self.numVars = 0 # number of variables
    self.variables = [] # names of variables
    self.values = {} # domains for each variable (dictionary)
    self.unaryFactors = {} # unary constraints
    self.binaryFactors = {} # binary constraints, explicit representation

    def add_variable (self, var, domain)
    def get_neighbor_vars (self, var)
    def add_unary_factor (self, var, factorFunc)
    def add_binary_factor (self, var1, var2, factor_func)

}
```

Submission.py


```
class BacktrackingSearch{
    def solve(self, csp, mcv = False, ac3 = False)
    def backtrack(self, assignment, numAssigned, weight)
    def get_unassigned_variable(self, assignment)
    def arc_consistency_check(self, assignment, var):
}
```

Pseudo-code for Backtracking


```
def Backtracking (assignment){

    if all variables assigned:
        add current assignment to assignments
    else:
        var <- choose an unassigned variable

    for value in domain of var:
        add{var = value} to assignment
        Backtracking(assignment)
        delete var from assignment
}
```

Pseudo-code for Backtracking


```
def Backtracking:
    if all variables assigned:
        add current assignment to assignments
    else:
        if MRC not implemented:
            randomly choose var from unassigned variables
        else(MRC implemented):
            choose variable by MCV
        for value in domain of var:
            add{var, value} to assignment
            if AC-3 not implemented:
                Backtracking(assignment)
            else(AC-3 implemented):
                localCopy <- store a deep copy for domains
                arc consistency checking
                Backtracking(assignment)
                use pre-stored localCopy to undo modify on domain
            delete assigned value for var
}
```

Problem b —— Most Constrained Variable

- **Select a variable with the least number of remaining domain**

```
Hints:

• self.csp.variables gives you all the variables in csp

• given var, self.domains[var] gives you all the possible
  values

• get_delta_weight(self, assignment, var, val) return the
  change of weights after assign val to var. In this case
  , will return 0 for inconsistent, 1 for consistent with
  assignment

• assignment, a variable, and a proposed value to this
  variable
```

Problem c —— AC-3

- After assigning new value to var, update domain for var's neighbors with AC-3.

```
Hint: How to get variables neighboring variable |var|?
=> for var2 in self.csp.get_neighbor_vars(var):
       # use var2

Hint: How to check if a value or two values are inconsistent?
- For unary factors
  => self.csp.unaryFactors[var1][val1] == 0

- For binary factors

  => self.csp.binaryFactors[var1][var2][val1][val2] == 0
```