AI

Unit – 10
Game Playing

By :- Dr. Maulik Dhamecha

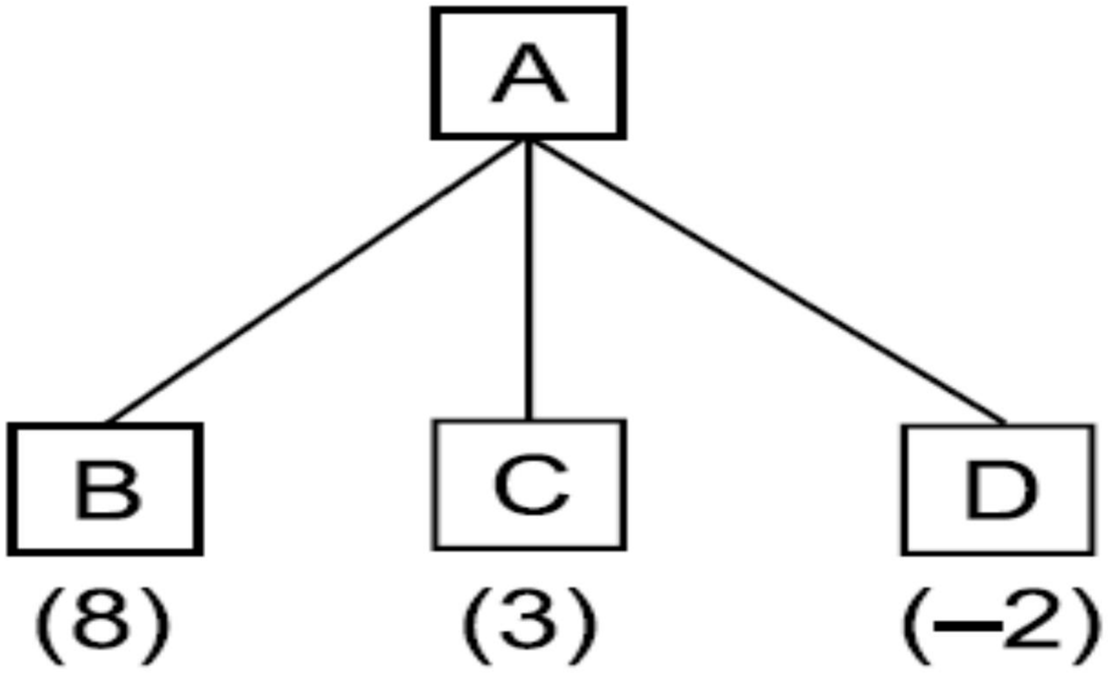
Overview

- A game provides a structured task in which it is very easy to measure success or failure.
- Consider chess. The average branching factor is around 35. in an average game, each player might make 50 moves. So in order to examine the complete game free, we would have to examine 35^{100} positions.
- Depth-limited search.
- Static evaluation function.
- Credit assignment problem.
- Minimax search algorithm.

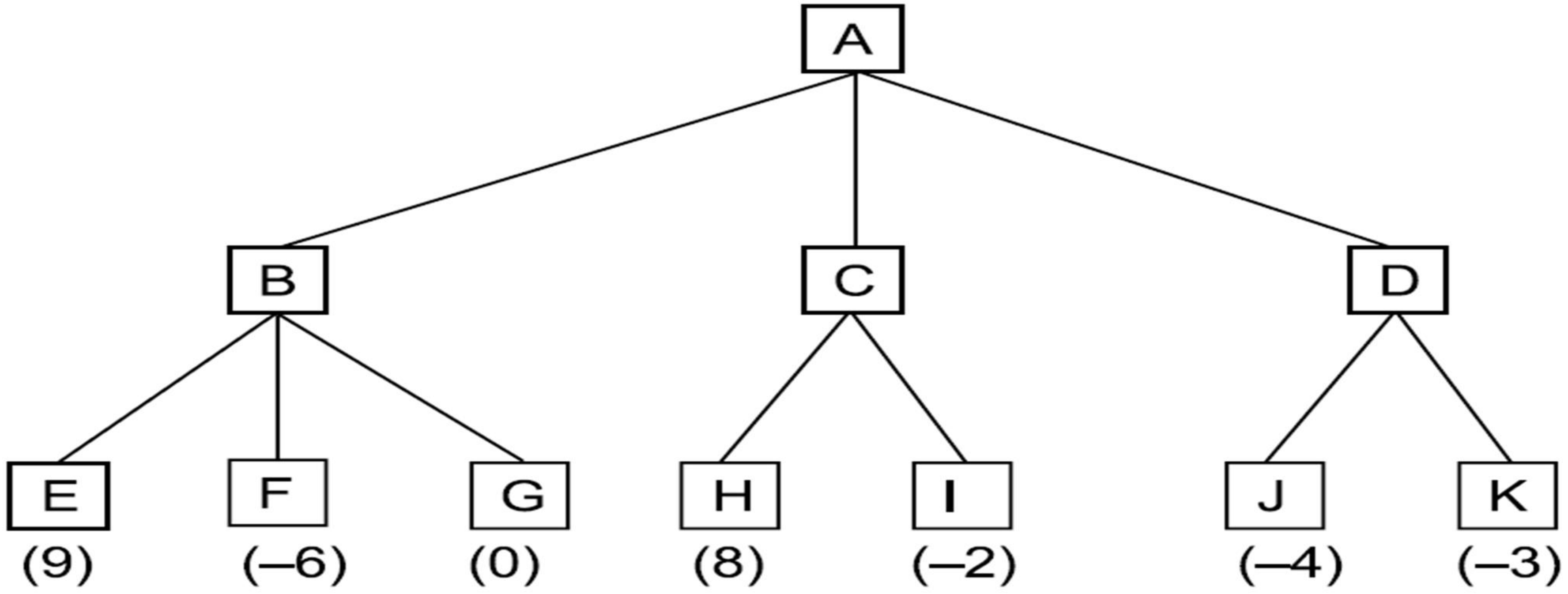
Overview

- In order to determine the best or optimal solution, consider the following components.
- Initial state : initial situation of board
- Expand function : use full to build all successor states
- Cost function : determine cost of the state
- Goal State : state with maximum cost

One – Ply Search



Two – Ply Search



Minimax Search

- The minimax search is a depth first and depth limited procedure.
- The idea is to start at the current position and use the posible-move generator to generate the set of possible successor positions.
- Now we can apply the static evolution function to those positions and simply choose the best one.
- After doing so, we can back that value up to the starting position to represent our evolution of it.

Minimax Search

- Here we assume that static evolution function returns larger values to indicate good situations for us.
- So our goal is to maximize the value of the static evaluation function of the next board position.
- The opponents’ goal is to minimize the value of the static evaluation function.
- ***The alternation of maximizing and minimizing at alternate play when evaluations are to be pushed back up corresponds to the opposing strategies of the two players is called MINIMAX.***

Minimax Search

Algorithm : MINIMAX (Position, Depth, Player)

1.If DEEP-ENOUGH(Position, Depth), then return the structure

    VALUE = STATIC(Position, Player);
     PATH = nil

This indicates that there is no path from this node and that its value is that determined by the static evaluation function.

2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(Position Player) and setting SUCCESSORS to the list it returns.

3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned if DEEP-ENOUGH had returned true.

4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows. Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the best score that can be achieved by an element of SUCCESSORS. For each element SUCC of SUCCESSORS, do the following:

Minimax Search (Cont'd)

(a)Set RESULT-SUCC to

M1NIMAX(SUCC, Depth + 1, OPPOSITE(Player))
This recursive call to MINIMAX will actually carry out the exploration of SUCC.

(b) Set NEW-VALUE to - VALUE(RESULT-SUCC). This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level.

(c) If NEW-VALUE > BEST-SCORE, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:

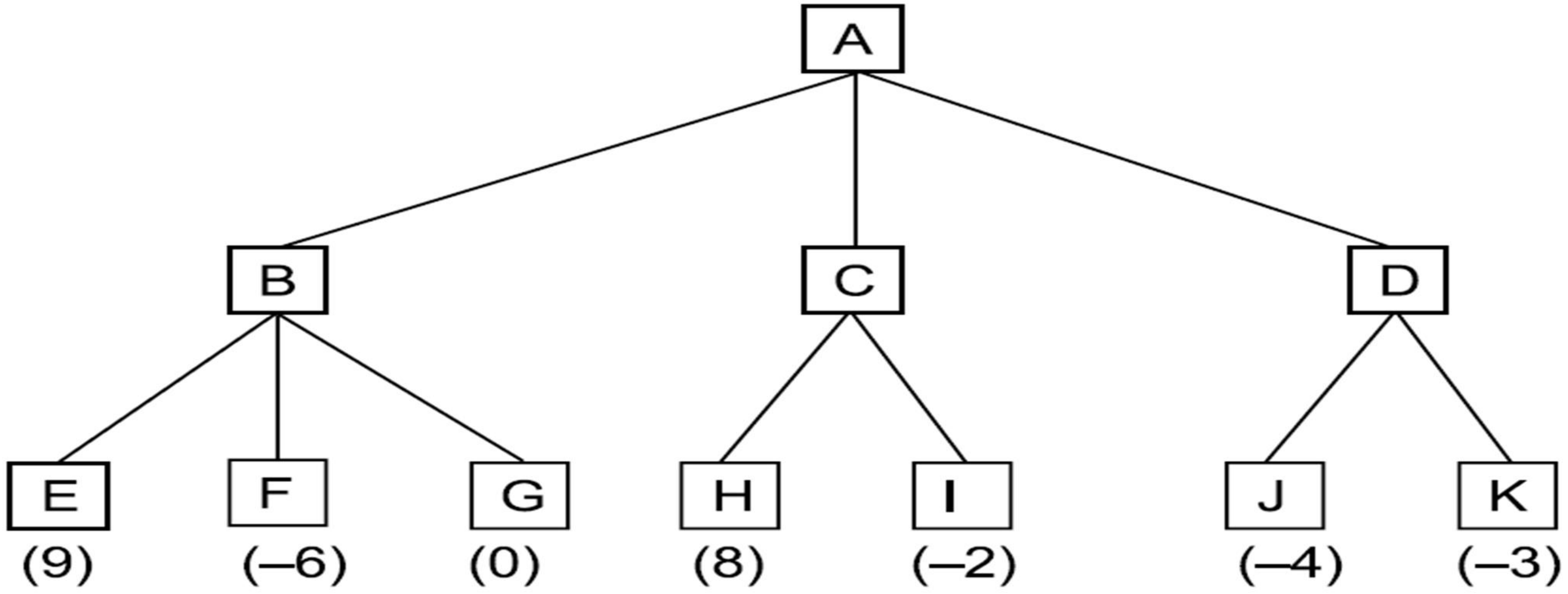
(i) Set BEST-SCORE to NEW-VALUE.

(ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined by the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).

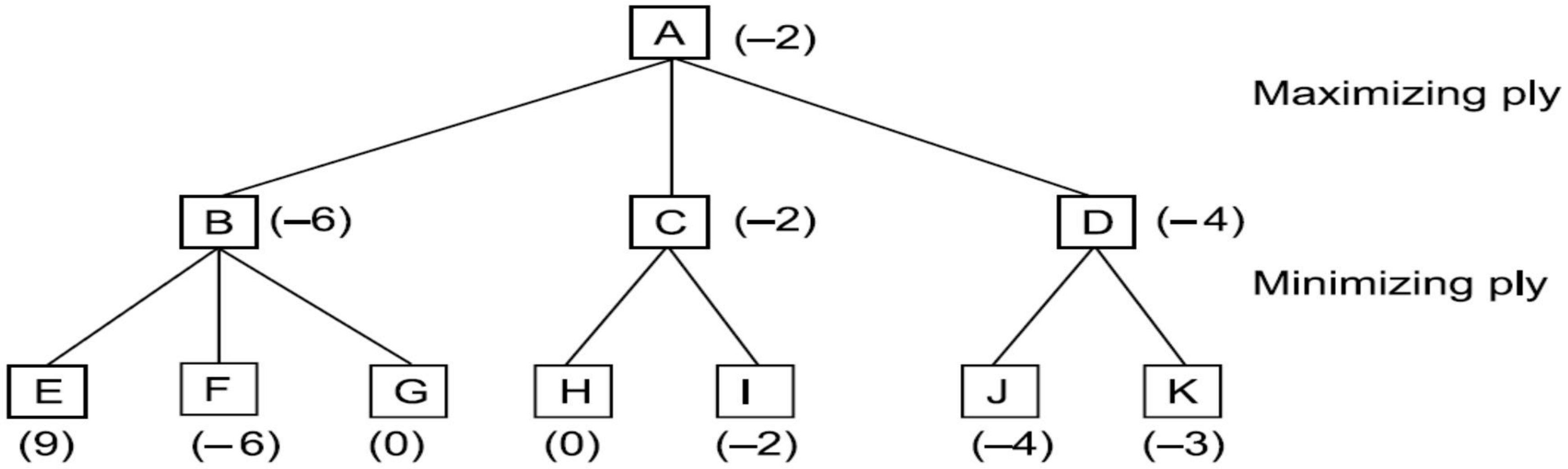
5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So return the structure

VALUE = BEST-SCORE
PATH = BEST-PATH

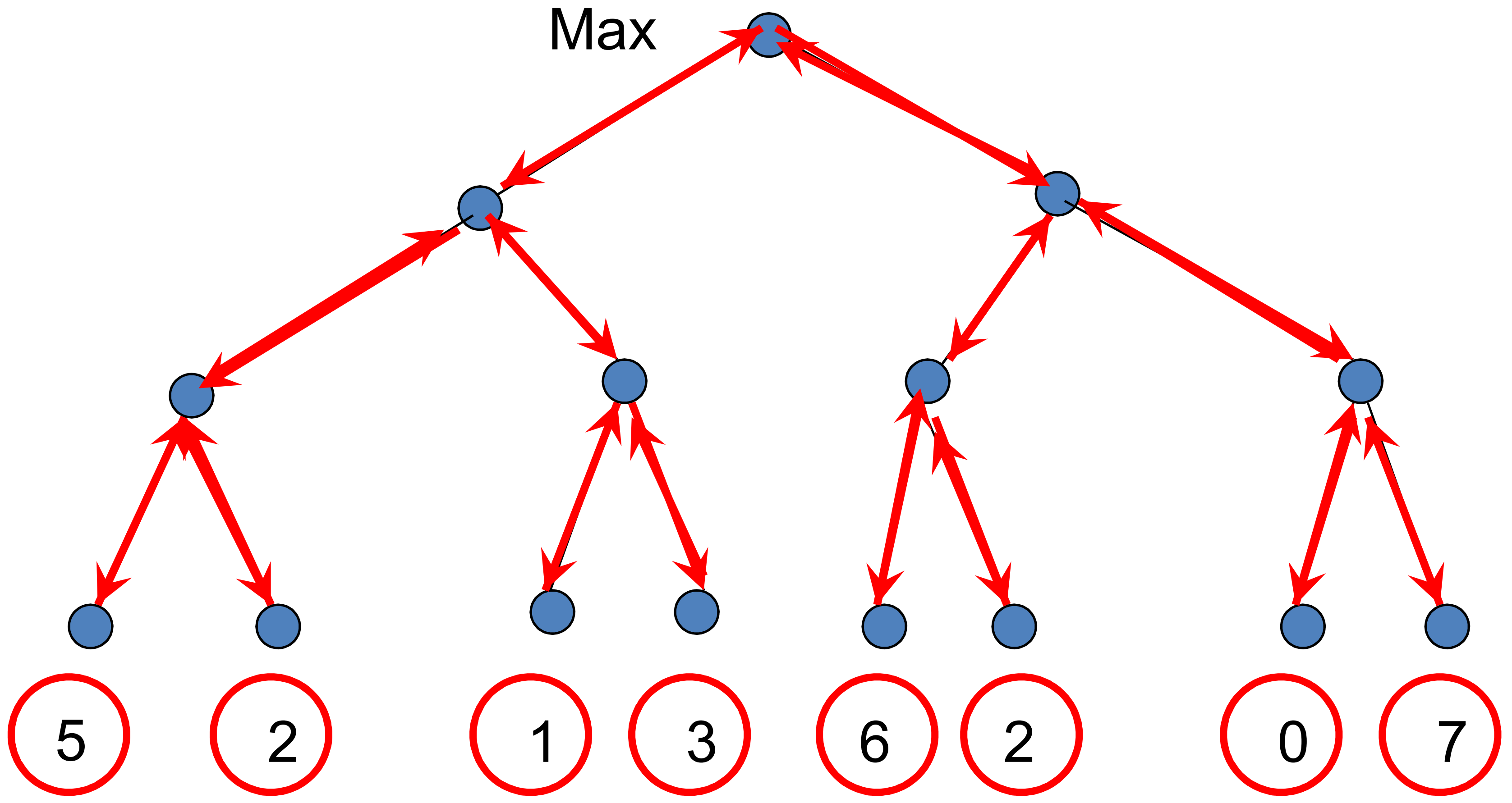
Two – Ply Search



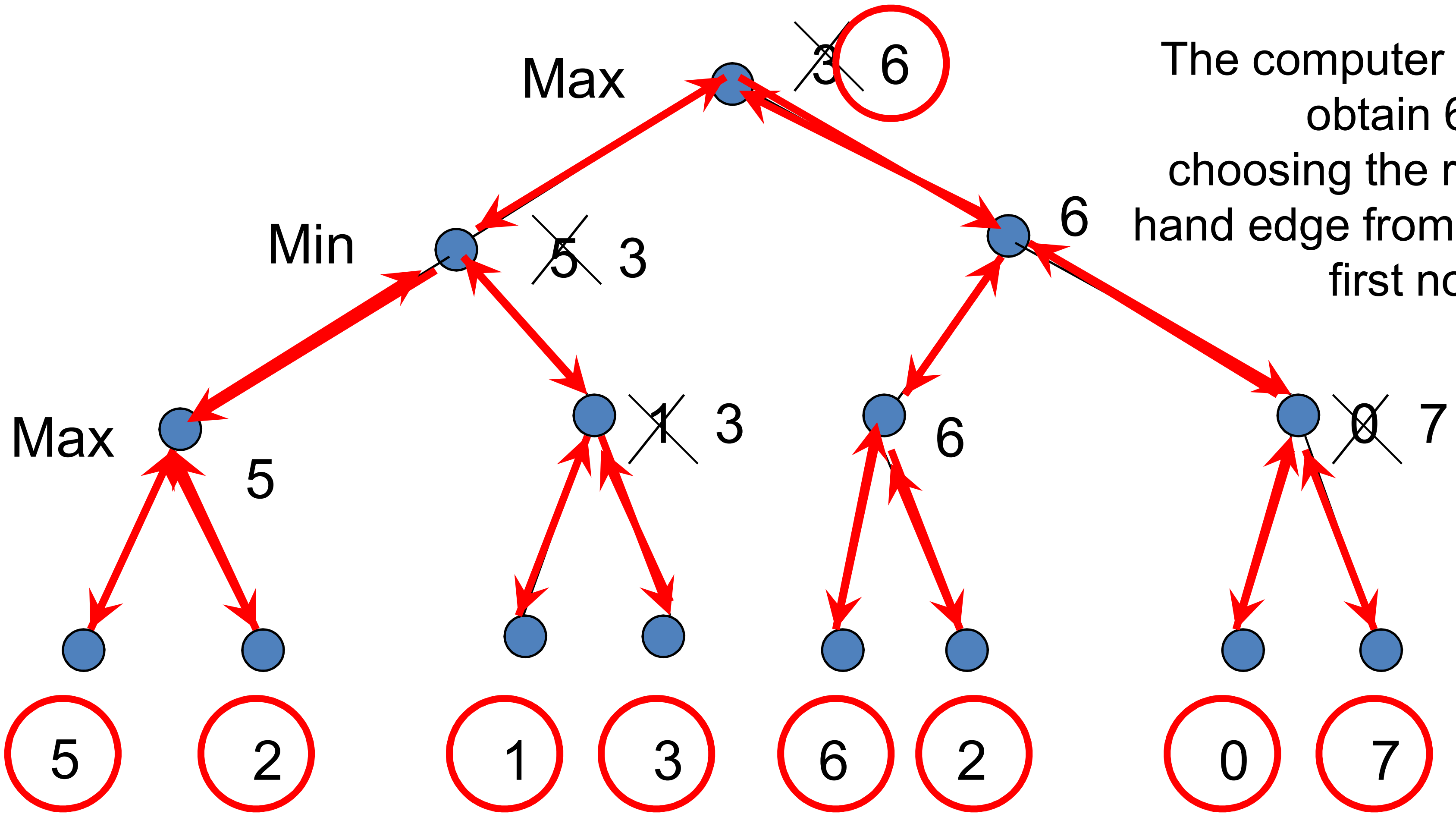
Backing Up the Values of a Two – Ply Search



Minimax Example



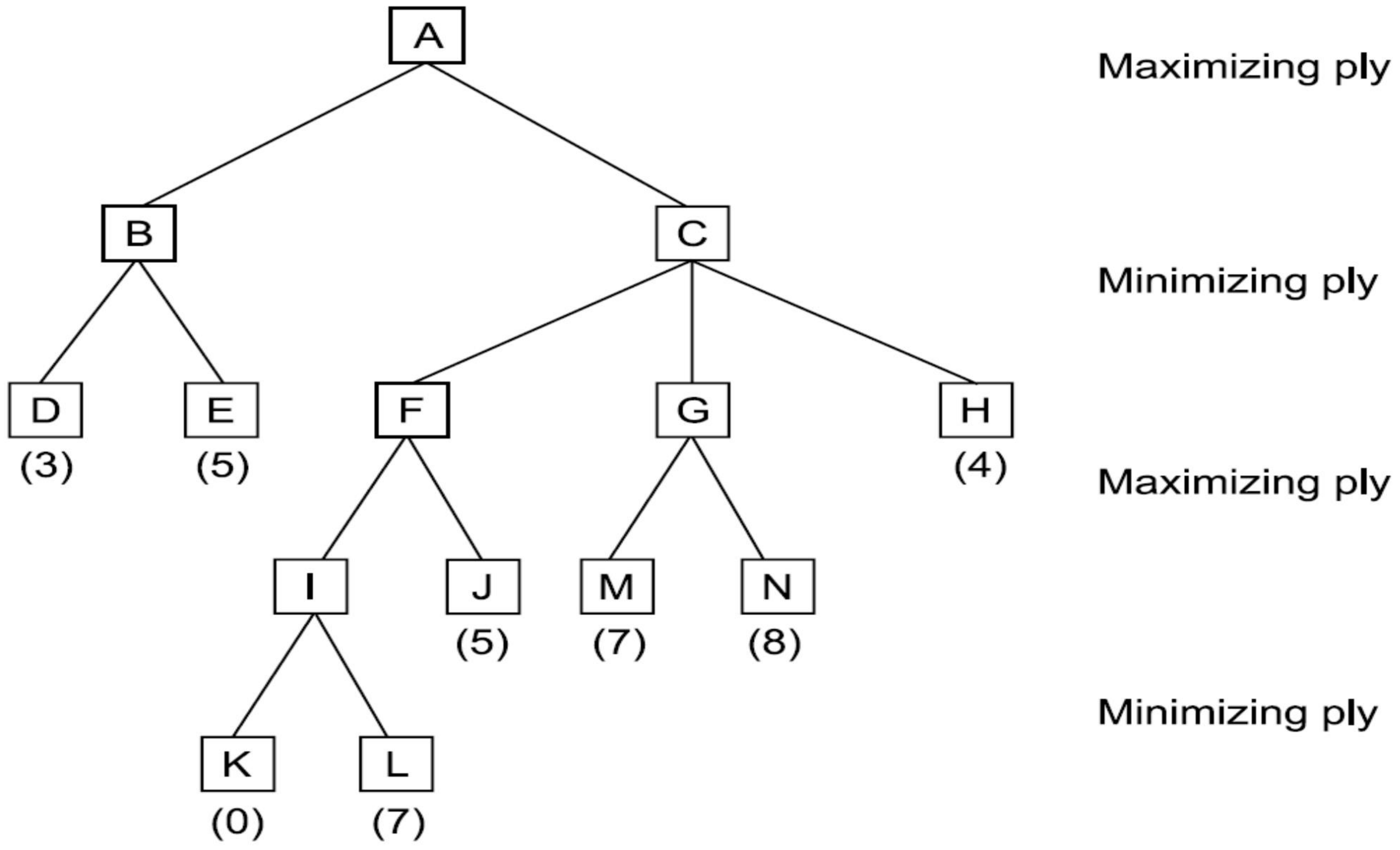
Minimax Example



The computer can obtain 6 by choosing the right hand edge from the first node.

Alpha – Beta Pruning

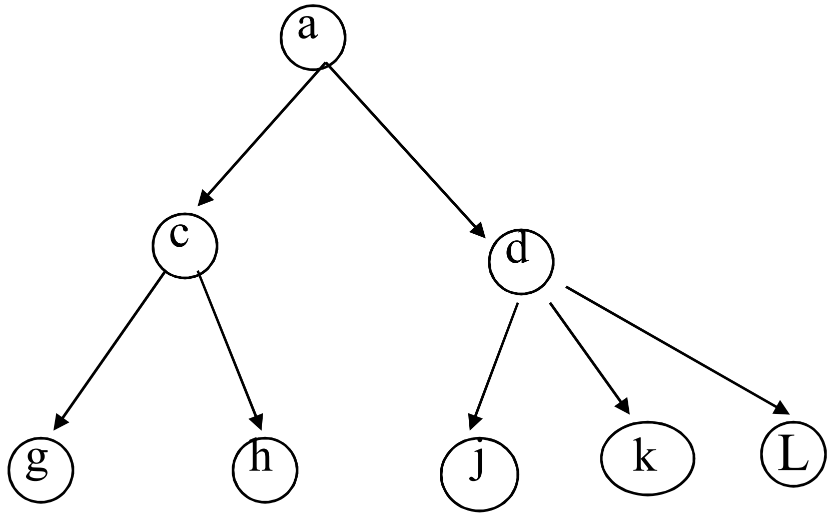
Alpha and Beta Cutoffs

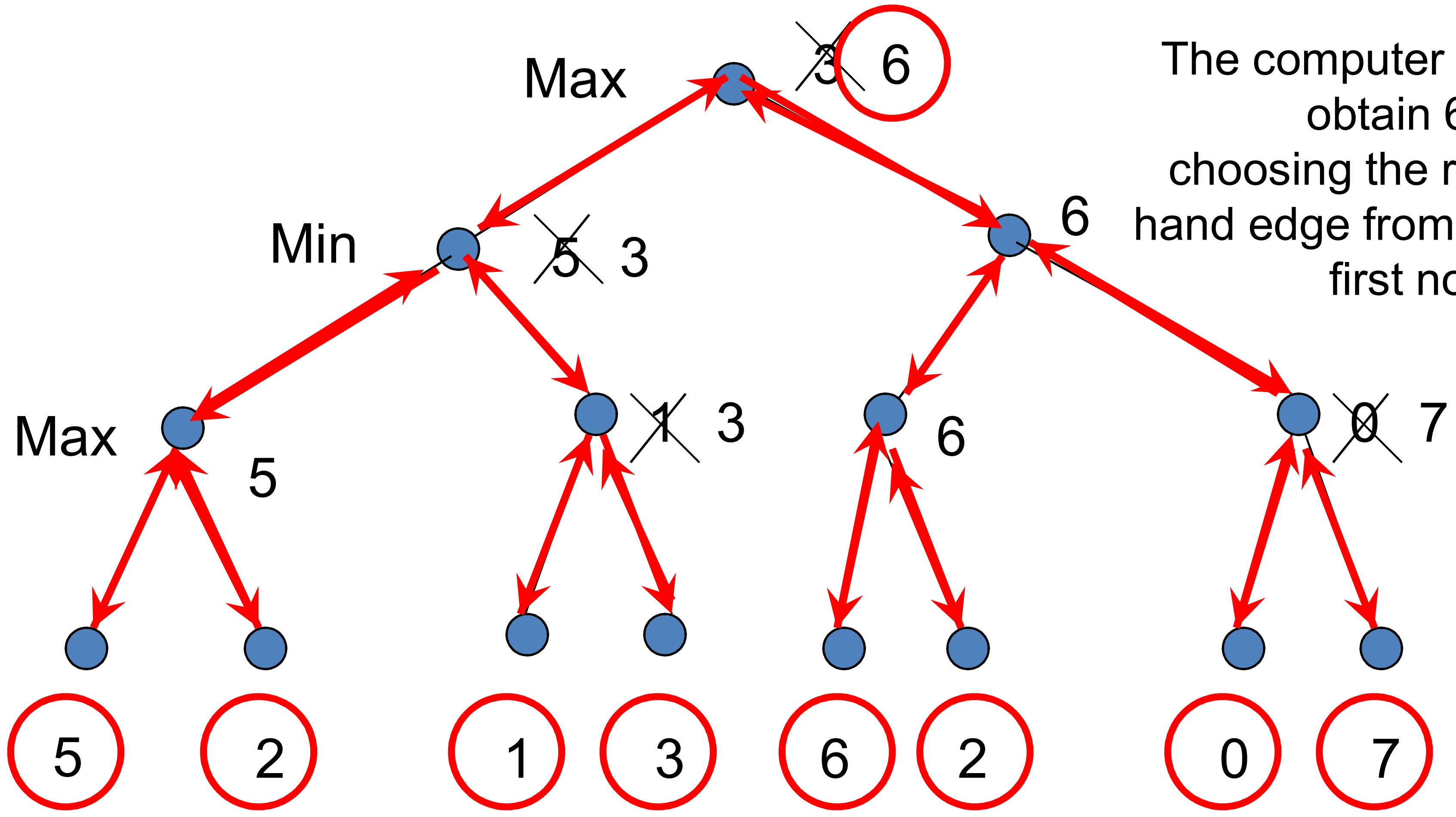


Additional Refinements

- Waiting for Quiescence
- Secondary search
- Using book moves
- Alternatives to MINIMAX

Iterative Deepening Search

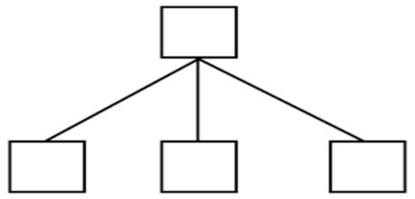




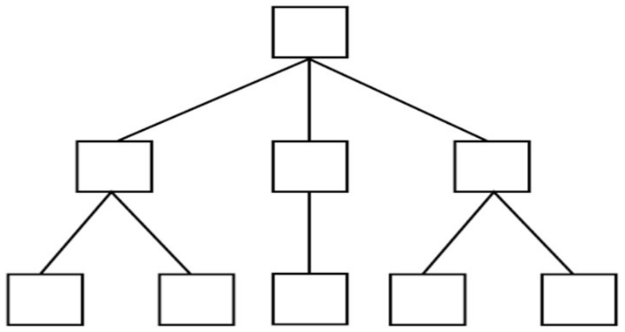
The computer can obtain 6 by choosing the right hand edge from the first node.

Exercise

Iterative Deepening



Iteration 1.



Iteration 2.

Planning

Components of Planning a System

- Choose the best rule to apply next based on the best available heuristic information.
- Apply the chosen rule to compute the new problem state that arises from its application.
- Detect when a solution has been found.
- Detect dead ends so that they can be abandoned and the system’s effort directed in more fruitful directions.
- Detect when an almost correct solution has been found and employ special techniques to make it totally correct.

Components of a planning system

- Choose the best rule to apply next, based on the best available heuristic information.

➢ The most widely used technique for selecting appropriate rules to apply is first to isolate a set of differences between desired goal state and then to identify those rules that are relevant to reduce those differences.

➢ If there are several rules, a variety of other heuristic information can be exploited to choose among them.

Components of a planning system

- Apply the chosen rule to compute the new problem state that arises from its application.

➢ In simple systems, applying rules is easy. Each rule simply specifies the problem state that would result from its application.

➢ In complex systems, we must be able to deal with rules that specify only a small part of the complete problem state.

➢ One way is to describe, for each action, each of the changes it makes to the state description.

Components of a planning system

- Detect when a solution has been found.

➢ A planning system has succeeded in finding a solution to a problem when it has found a sequence of operators that transforms the initial problem state into the goal state.

➢ How will it know when this has been done?

➢ In simple problem-solving systems, this question is easily answered by a straightforward match of the state descriptions.

➢ One of the representative systems for planning systems is predicate logic. Suppose that as a part of our goal, we have the predicate P(x).

➢ To see whether P(x) is satisfied in some state, we ask whether we can prove P(x) given the assertions that describe that state and the axioms that define the world model.

Components of a planning system

- Detect dead ends so that they can be abandoned and the system’s effort directed in more fruitful directions.

➢ As a planning system is searching for a sequence of operators to solve a particular problem, it must be able to detect when it is exploring a path that can never lead to a solution.

➢ The same reasoning mechanisms that can be used to detect a solution can often be used for detecting a dead end.

➢ If the search process is reasoning forward from the initial state, it can prune any path that leads to a state from which the goal state cannot be reached.

➢ If search process is reasoning backward from the goal state, it can also terminate a path either because it is sure that the initial state cannot be reached or because little progress is being made.

Components of a planning system

- Detect when an almost correct solution has been found and employ special techniques to make it totally correct.

➢ The kinds of techniques discussed are often useful in solving nearly decomposable problems.

➢ One good way of solving such problems is to assume that they are completely decomposable, proceed to solve the sub-problems separately, and then check that when the sub-solutions are combined, they do in fact give a solution to the original problem.

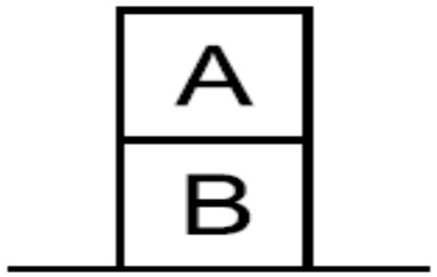
The Blocks Word

- **Operators :**
  - **UNSTACK(A, B)** — Pick up block A from its current position from block B. The arm must be empty and block A must have no blocks on top of it.
  - **STACK(A, B)** — Place block A on block B. The arm must already be holding block A and the surface of B must be clear.
  - **PICKUP(A)** — Pick up block A from the table and hold it. The arm must be empty and there must be nothing on top of block A.
  - **PUTDOWN(A)** — Put block A down on the table. The arm must have been holding block A.

The Blocks World (Cont’d)

- **Predicates :**
  - ON(A, B)—Block A is on block B.
  - ONTABLE(A)—Block A is on the table.
  - CLEAR(A)—There is nothing on top of block A.
  - HOLDING(A)—The arm is holding block A.
  - ARMEMPTY—The arm is holding nothing.
- **Inference rules :**
  - $[\exists x : \mathrm{HOLDING}(x)] \rightarrow \neg\mathrm{ARMEMPTY}$
  - $\forall x : \mathrm{ONTABLE}(x) \rightarrow \neg\exists y : ON(x,y)$
  - $\forall x : [\neg\exists y : ON(y, x)] \rightarrow \mathrm{CLEAR}(x)$

A Simple Blocks World Description



$ON(A, B, S0) \wedge$
$ONTABLE (B, S0) \wedge$
$CLEAR (A, S0)$

STRIPS –Style Operators for the Blocks World

STACKS(x,y)

P: CLEARC(y) $\wedge$ HOLDING(x)
D: CLEAR(y) $\wedge$ HOLDING(x)
A: ARMEMPTY $\wedge$ ON(x, y)

UNSTACK(x,y)

P: ON(x, y) $\wedge$ CLEAR(x) $\wedge$ ARMEMPTY
D: ON(x, y) $\wedge$ ARMEMPTY
A: HOLDINGS(x) $\wedge$ CLEARC(y)

PICKUP(x)

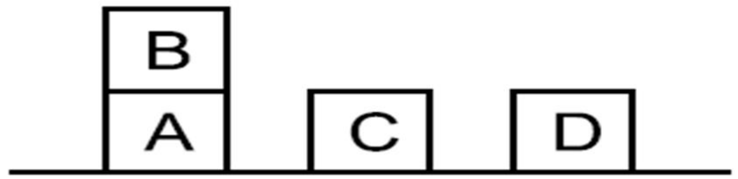
P: CLEAR(x) $\wedge$ ONTABLE(x) $\wedge$ ARMEMPTY
D: ONTABLE(x) $\wedge$ ARMEMPTY
A: HOLDING(x)

PUTDOWN(x)

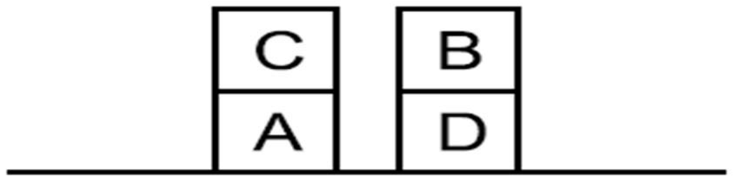
P: HOLDING(x)
D: HOLDING(x)
A: ONTABLE(x) $\wedge$ ARMEMPTY

A Simple Search Tree

A Very Simple Blocks World Problem



start: ON(B, A) $\wedge$
ONTABLE(A) $\wedge$
ONTABLE(C) $\wedge$
ONTABLE(D) $\wedge$
ARMEMPTY



goal: ON(C, A) $\wedge$
ON(B, D) $\wedge$
ONTABLE(A) $\wedge$
ONTABLE(D)

Goal Stack Planning

- Initial goal stack :

  ON(C, A) ∧ ON(B, D) ∧ ONTABLE(A) ∧ ONTABLE(D)

- Choose to work on ON(C, A) before ON(B, D):

  ON(C, A)
  ON(B, D)
  ON(C, A) ∧ ON(B, D) ∧ OTAD

- Achieve (ON(C, A) with STACK(C, A):

  STACK(C, A)
  ON(B, D)
  ON(C, A) ∧ ON(B, D) ∧ OTAD

- Add STACK'S preconditions:

  CLEAR(A)
  HOLDING(C)
  CLEAR(A) ∧ HOLDING(C)
  STACK(C, A)
  ON(B, D)
  ON(C, A) ∧ ON(B, D) ∧ OTAD

Goal Stack Planning (Cont’d)

- Achieve CLEAR(A) with UNSTACK (B, A) :

  ON(B, A)
  CLEAR(B)
  ARMEMPTY
  ON(B, A) $\wedge$ CLEAR(B) $\wedge$ ARMEMPTY
  UNSTACK(B, A)
  HOLDING(C)
  CLEAR(A) $\wedge$ HOLDING(C)
  STACK(C, A)
  ON(B, D)
  ON(C, A) $\wedge$ ON(B, D) $\wedge$ OTAD

- Pop satisfied predicates :

  HOLDING(C)
  CLEAR(A) $\wedge$ HOLDING(C)
  STACK(C, A)
  ON(B, D)
  ON(C, A) $\wedge$ ON(B, D) $\wedge$ OTAD

Goal Stack Planning (Cont'd)

- Achieve HOLDING(C) with UNSTACK(C,x) :

```
ON(C, x)
CLEAR(C)
ARMEMPTY
ON(C, x) ∧ CLEAR(C) ∧
        ARMEMPTY
UNSTACK(C, x)
CLEAR(A) ∧ HOLDING(C)
STACK(C, A)
ON(B, D)
ON(C, A) ∧ ON(B, D) ∧ OTAD
```

Goal Stack Planning (Cont’d)

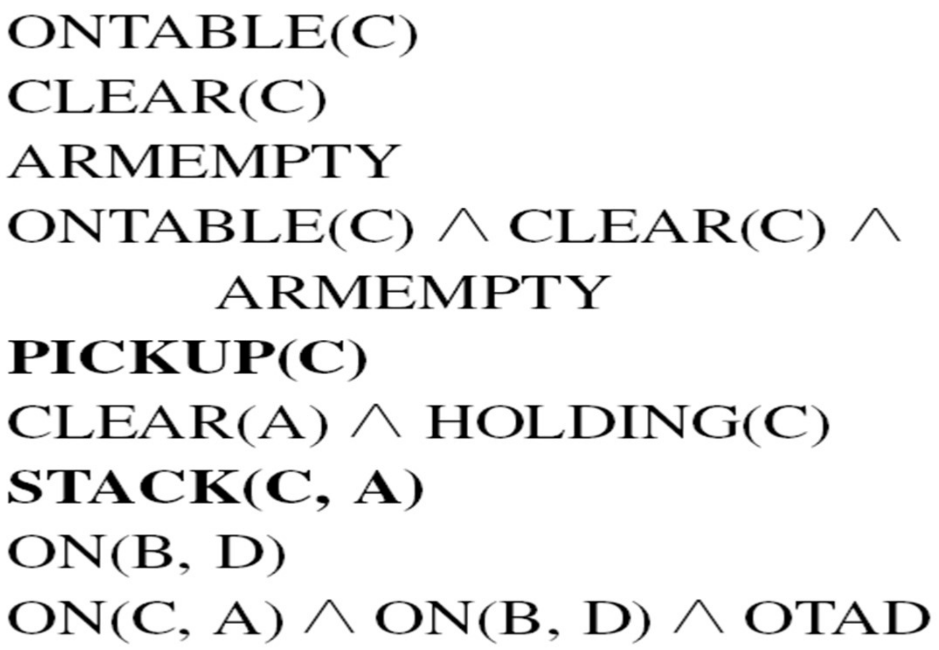
- Achieve ON(C,x) by STACK(C,x) :

CLEAR(*x*)
HOLDING(C)
CLEAR(*x*) $\wedge$ HOLDING(C)
STACK(C, *x*)
CLEAR(C)
ARMEMPTY
ON(C, *x*) $\wedge$ CLEAR(C) $\wedge$ ARMEMPTY
UNSTACK(C, *x*)
CLEAR(A) $\wedge$ HOLDING(C)
STACK(C, A)
ON(B, D)
ON(C, A) $\wedge$ ON(B, D) $\wedge$ OTAD

- Terminate path because **HOLDING(C)** is duplicated.

Goal Stack Planning (Cont’d)

- Achieve HOLDING(C) with PICUP, not UNSTACK:



```
ONTABLE(C)
CLEAR(C)
ARMEMPTY
ONTABLE(C) ∧ CLEAR(C) ∧
        ARMEMPTY
PICKUP(C)
CLEAR(A) ∧ HOLDING(C)
STACK(C, A)
ON(B, D)
ON(C, A) ∧ ON(B, D) ∧ OTAD
```

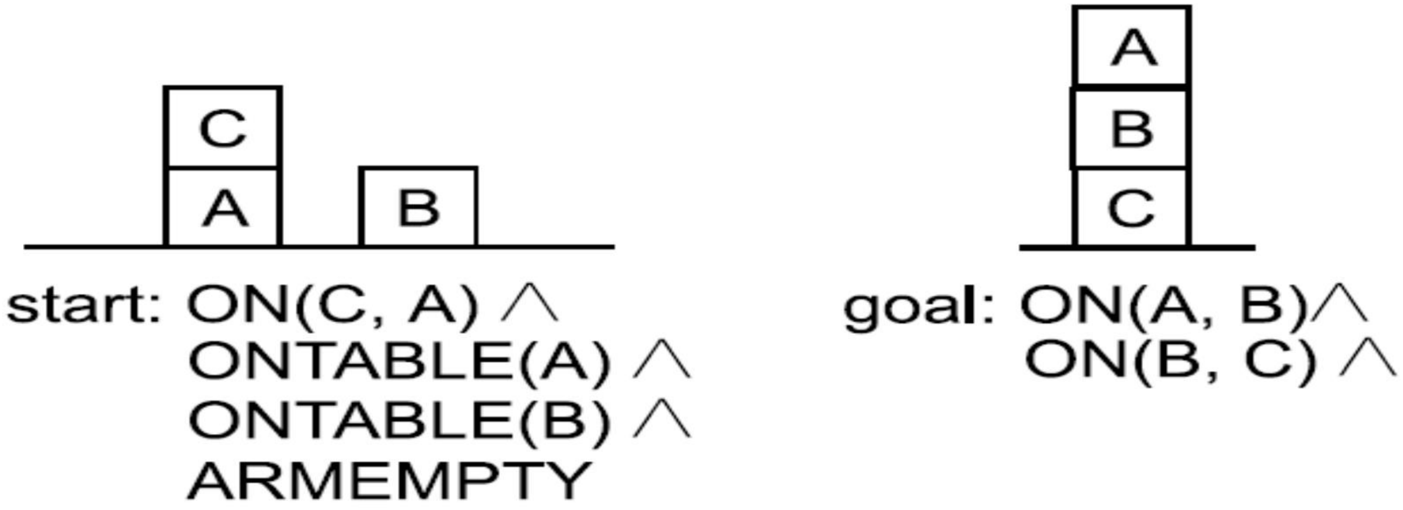
Goal Stack Planning (Cont'd)

- Pop ONTABLE(C) and CLEAR(C), and achieve ARMEMPTY by STACK(B, D) :

  CLEAR(D)
  HOLDING(B)
  CLEAR(D) $\wedge$ HOLDING(B)
  STACK(B, D)
  ONTABLE(C) $\wedge$ CLEAR(C) $\wedge$ ARMEMPTY
  PICKUP(C)
  CLEAR(A) $\wedge$ HOLDING(C)
  STACK(C, A)
  ON(B,D).
  ON(C, A) $\wedge$ ON(B, D) $\wedge$ OTAD

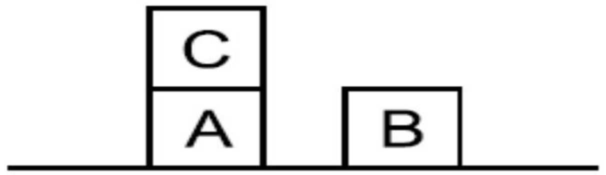
- Pop entire stack, and return plan :
  1. UNSTACK(B.A)
  2. STACK(B,D)
  3. PICKUP(C)
  4. STACK(C, A)

Nonlinear Planning

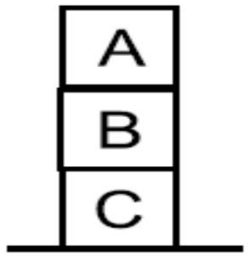


- Begin work on the goal ON(A, B) by clearing A, thus putting C on the table.
- Achieve the goal ON(B, C) by stacking B on C.
- Complete the goal ON(A, B) by stacking A on B.

A Slightly Harder Blocks Problem



start: $ON(C, A) \wedge$
$ONTABLE(A) \wedge$
$ONTABLE(B) \wedge$
$ARMEMPTY$



goal: $ON(A, B) \wedge$
$ON(B, C) \wedge$

Nonlinear Planning using Constraint Posting

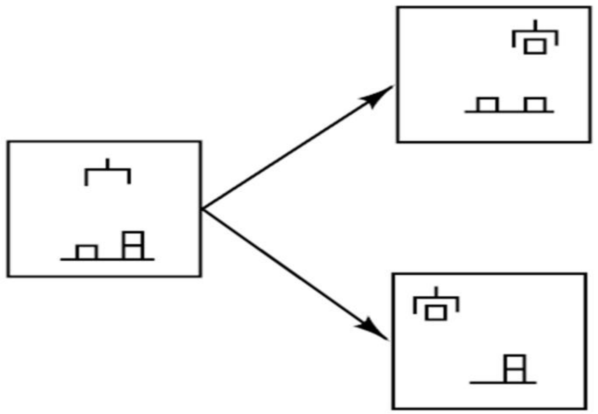
- Difficult problems cause goal interactions.
- The operators used to solve one sub-problem may interfere with the solution to a previous sub-problem.
- Most problems require an intertwined plan in which multiple sub-problems are worked on simultaneously.
- Such a plan is called nonlinear plan because it is not composed of a linear sequence of complete sub-plans.

Constraint Posting

- The idea of constraint posting is to build up a plan by incrementally hypothesizing operators, partial orderings between operators, and binding of variables within operators.
- At any given time in the problem-solving process, we may have a set of useful operators but perhaps no clear idea of how those operators should be ordered with respect to each other.
- A solution is a partially ordered, partially instantiated set of operators to generate an actual plan, and we convert the partial order into any number of total orders.

Constraint Posting versus State Space Search

State Space Search

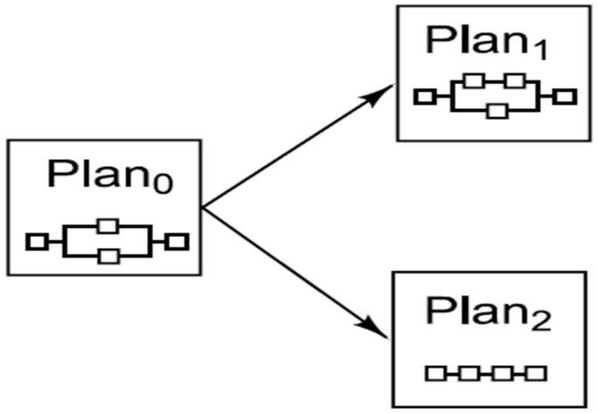


Moves in the space:
- Modify world state via operator

Model of time:
- Depth of node in search space

Plan stored in:
- Series of state transitions

Constraint Posting Search



Move in the space:
- Add operators
- Order operators
- Bind variabls
- Or otherwise constrain plan

Model of time:
- Partially ordered set of operators

Plan stored in:
- Single node

Heuristics for Planning Using Constraint Posting (TWEAK)

1. Step Addition—Creating new steps for a plan.
2. Promotion—Constraining one step to come before another in a final plan.
3. Declobbering—Placing one (possibly new) step s_2 between two old steps s_1 and s_3, such that s_2 reasserts some precondition of s_3 that was negated (or "clobbered") by s_1.
4. Simple Establishment—Assigning a value to a variable, in order to ensure the preconditions of some step.
5. Separation—Preventing the assignment of certain values to a variable.

Constructing a Nonlinear Plan

- Achieve (ON(A, B) and ON(B, C) by step addition :

```
    CLEAR(B)                 CLEAR(C)
  * HOLDING(A)             * HOLDING(B)
  ____________             ____________
    STACK(A,B)               STACK(B,C)
  ____________             ____________
    ARMEMPTY                 ARMEMPTY
    ON(A,B)                  ON(B,C)
    ¬CLEAR(B)                ¬CLEAR(C)
    ¬HOLDING(A)              ¬HOLDING(B)
```

- Achieve (HOLDING(A), HOLDING(B) by step addition :

```
  * CLEAR(A)               * CLEAR(B)
    ONTABLE(A)               ONTABLE(B)
  * ARMEMPTY               * ARMEMPTY
    PICKUP(A)                PICKUP(B)
  ¬ ONTABLE(A)             ¬ ONTABLE(B)
  ¬ ARMEMPTY               ¬ ARMEMPTY
    HOLDING(A)               HOLDING(B)
```

Constructing a Nonlinear Plan (Cont'd)

- Ensure holding preconditions with ordering constraints :

  $\text{PICKUP(A)} \leftarrow \text{STACK(A, B)}$
  $\text{PICKUP(B)} \leftarrow \text{STACK(B, C)}$

- Achieve CLEAR(B) with promotion :

  $\text{PICKUP(B)} \leftarrow \text{STACK(A, B)}$

- Achieve ARMEMPTY with promotion :

  $\text{PICKUP(B)} \leftarrow \text{PICKUP(A)}$

- Achieve other ARMEMPTY with declobbering :

  $\text{PICKUP(B)} \leftarrow \text{STACK(B, C)} \leftarrow \text{PICKUP(A)}$

Constructing a Nonlinear Plan (Cont’d)

- Achieve CLEAR(A) with step addition :

$$\begin{array}{l} *\ \mathrm{ON}(x,\mathrm{A}) \\ *\ \mathrm{CLEAR}(x) \\ *\ \mathrm{ARMEMPTY} \\ \hline \mathrm{UNSTACK}(x,\ \mathrm{A}) \\ \hline \neg\mathrm{ARMEMPTY} \\ \quad \mathrm{CLEAR}(\mathrm{A}) \\ \quad \mathrm{HOLDING}(\mathrm{A}) \\ \neg\mathrm{ON}(x,\ \mathrm{A}) \end{array}$$

- Achieve ON(x, A) with simple establishment :

$x = \mathrm{C}$ in step $\mathrm{UNSTACK}(x,\ \mathrm{A})$

- Achieve CLEAR(A) and ARMEMPTY with promotion :

$$\begin{array}{l} \mathrm{UNSTACK}(x,\ \mathrm{A}) \leftarrow \mathrm{STACK}(\mathrm{B},\ \mathrm{C}) \\ \mathrm{UNSTACK}(x,\ \mathrm{A}) \leftarrow \mathrm{PICKUP}(\mathrm{A}) \\ \mathrm{UNSTACK}(x,\ \mathrm{A}) \leftarrow \mathrm{PICKUP}(\mathrm{B}) \end{array}$$

Constructing a Nonlinear Plan (Cont’d)

- Re-achieve ARMEMPTY and PICKUP(B) by adding declobbering step :

HOLDING(C)

PUTDOWN(C)

¬HOLDING(C)
ONTABLE(x)
ARMEMPTY

UNSTACK(x, A) $\leftarrow$ PUTDOWN(C) $\leftarrow$ PICKUP(B)

- All preconditions satisfied, so return plan :

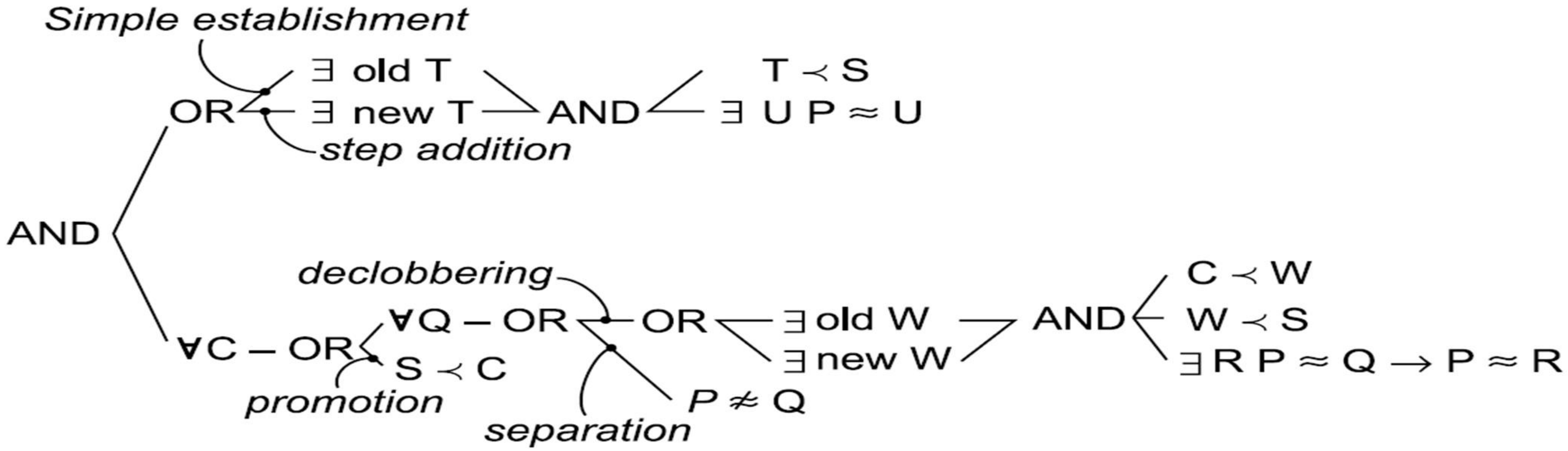
1. UNSTACK(C, A)
2. PUTDOWN(C)
3. PICKUP(B)
4. STACK(B, C)
5. PICKUP(A)
6. STACK(A, B)

TWEAK

Algorithm : Nonlinear Planning (TWEAK)

1. Initialize S to be the set of propositions in the goal state.
2. Remove some unachieved proposition P from S.
3. Achieve P by using step addition, promotion, declobbering, simple establishment, or separation.
4. Review all the steps in the plan, including any new steps introduced by step addition, to see if any of their preconditions are unachieved. Add to S the new set of unachieved preconditions.
5. If S is empty, complete the plan by converting the partial order of steps into a total order, and instantiate any variables as necessary.
6. Otherwise, go to step 2.

The Modal Truth Criterion for Telling whether Proposition P Necessarily Holds in State S



A Complex Operator

```
(OPERATOR
    (PRECONDITIONS
        (and(...)
            (forall (w ...)...)
            (not
                (exists ...)
                (or.....)))
    (POSTCONDITIONS
        (ADD (...))
        (DELETE (...))
        (if (and (...)(...))
            (ADD (...)(...))
            (DELETE (...)(...))))))
```

Hierarchical Planning

- In order to solve hard problems, a problem solver may have to generate long plans.
- It is important to be able to eliminate some of the details of the problem until a solution that addresses the main issues is found.
- Then an attempt can be made to fill in the appropriate details.
- Early attempts to do this involved the use of macro operators, in which larger operators were built from smaller ones.
- In this approach, no details were eliminated from actual descriptions of the operators.

ABSTRIPS

- A better approach was developed in ABSTRIPS systems which actually planned in a hierarchy of abstraction spaces, in each of which preconditions at a lower level of abstraction were ignored.
- The assignment of appropriate criticality value is crucial to the success of this hierarchical planning method.
- Those preconditions that no operator can satisfy are clearly the most critical.

- ABSTRIPS approach is as follows:

- First solve the problem completely, considering only preconditions whose criticality value is the highest possible.
- These values reflect the expected difficulty of satisfying the precondition.
- To do this, do exactly what STRIPS did, but simply ignore the preconditions of lower than peak criticality.
- Once this is done, use the constructed plan as the outline of a complete plan and consider preconditions at the next-lowest criticality level.
- Augment the plan with operators that satisfy those preconditions.
- Because this approach explores entire plans at one level of detail before it looks at the lower-level details of any one of them, it has been called length-first approach.

Example

– Solving a problem of moving robot, for applying an operator, PUSH-THROUGH DOOR, the precondition that there exist a door big enough for the robot to get through is of high criticality since there is nothing we can do about it if it is not true.

Reactive Systems

- The idea of reactive systems is to avoid planning altogether, and instead use the observable situation as a clue to which one can simply react.
- A reactive system must have access to a knowledge base of some sort that describes what actions should be taken under what circumstances.
- A reactive system is very different from the other kinds of planning systems we have discussed because it chooses actions one at a time.
- It does not anticipate and select an entire action sequence before it does the first thing.

Reactive Systems

- Example is a Thermostat. The job of the thermostat is to keep the temperature constant inside a room.
- Reactive systems are capable of surprisingly complex behaviors.
- The main advantage reactive systems have over traditional planners is that they operate robustly in domains that are difficult to model completely and accurately.
- Reactive systems dispense with modeling altogether and base their actions directly on their perception of the world.
- Another advantage of reactive systems is that they are extremely responsive, since they avoid the combinatorial explosion involved in deliberative planning.
- This makes them attractive for real time tasks such as driving and walking.

Other Planning Techniques

- Triangle tables : Provides a way of recording the goals that each operator is expected to satisfy as well as the goals that must be true for it to execute correctly.
- Meta-planning : A technique for reasoning not just about the problem being solved but also about the planning process itself.
- Macro-operators : Allow a planner to build new operators that represent commonly used sequences of operators.
- Case based planning: Re-uses old plans to make new ones.

End of Unit-10