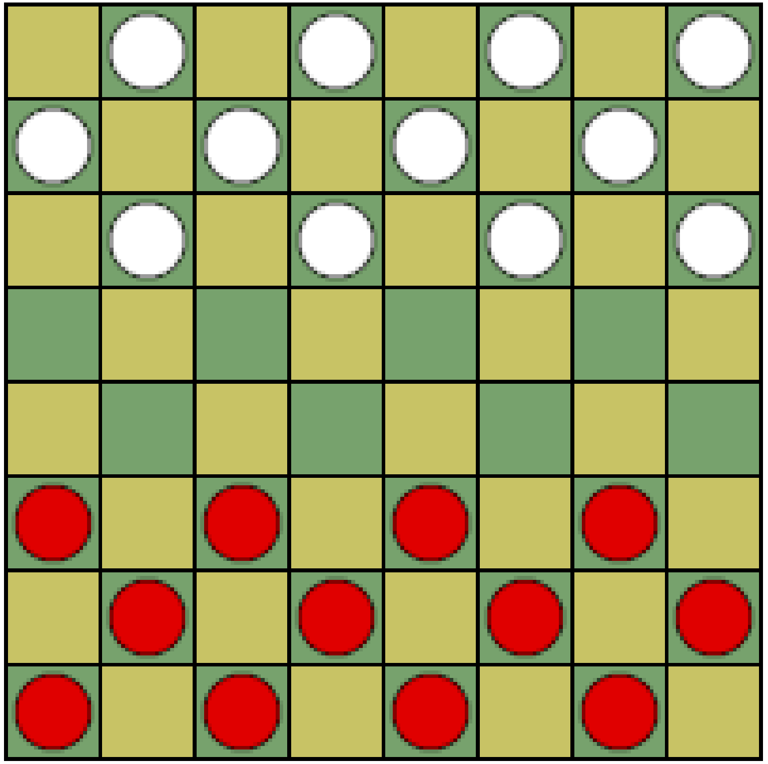
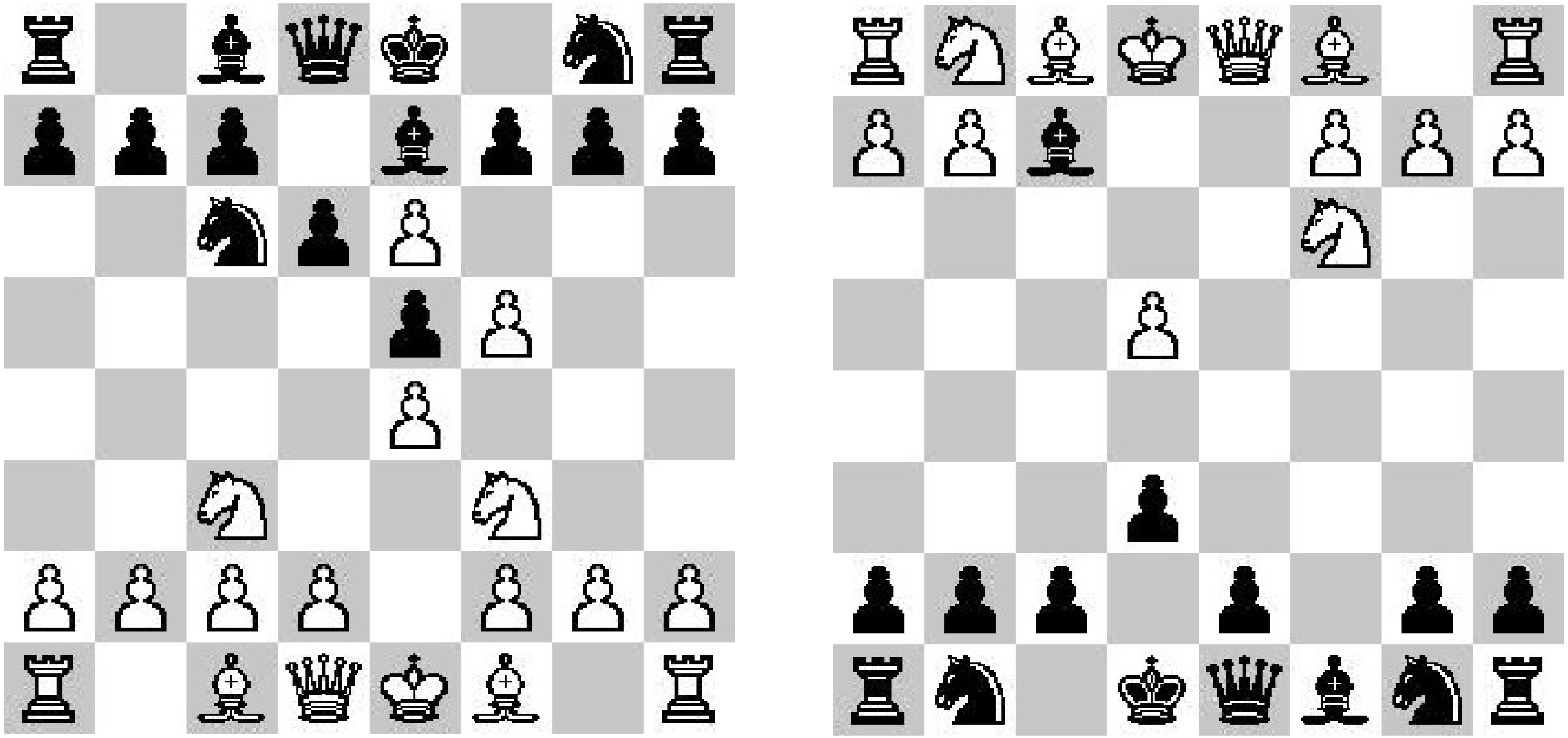
Lecture 4:
Incomplete Information

- Multiple Player Games and other Games with Incomplete Information
- State-Space Search with Multiple Players
- Minimax with α-β-Cutoff
- Planning under Uncertainty

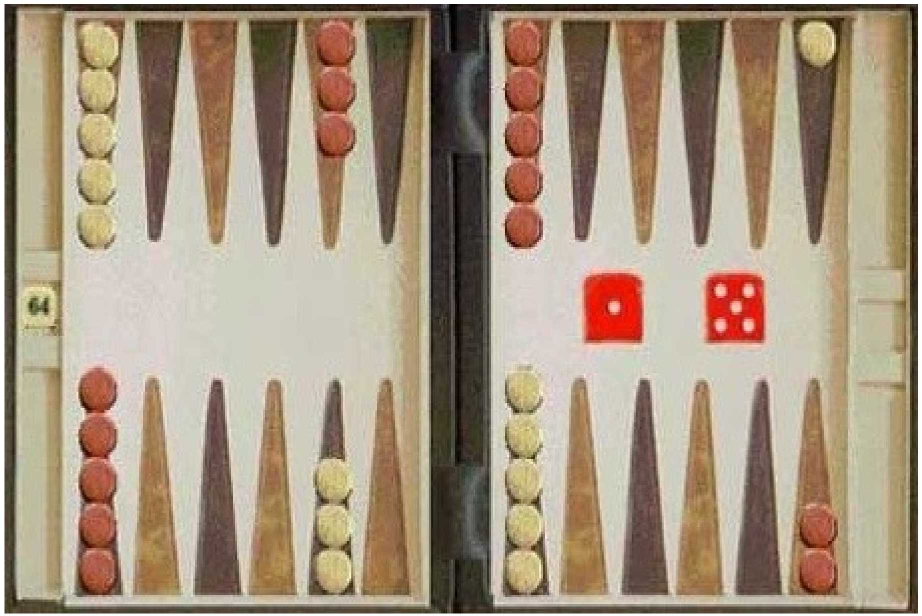
Checkers



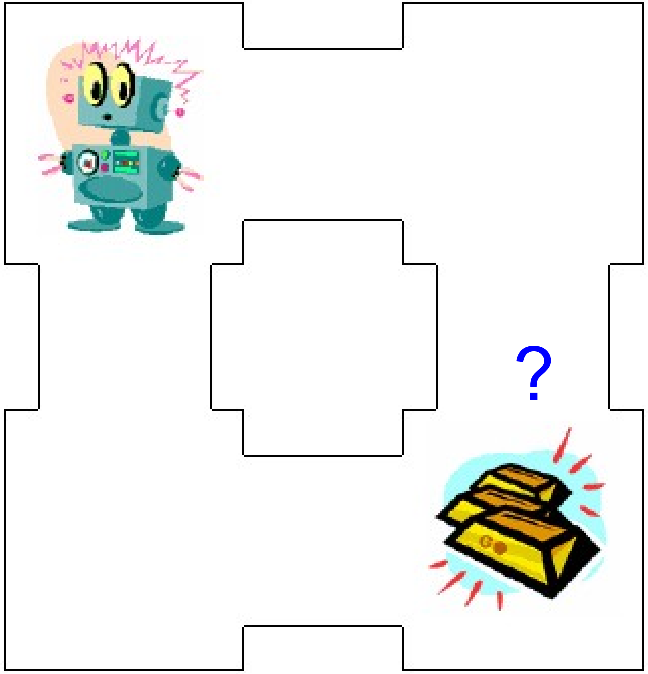
Bughouse Chess



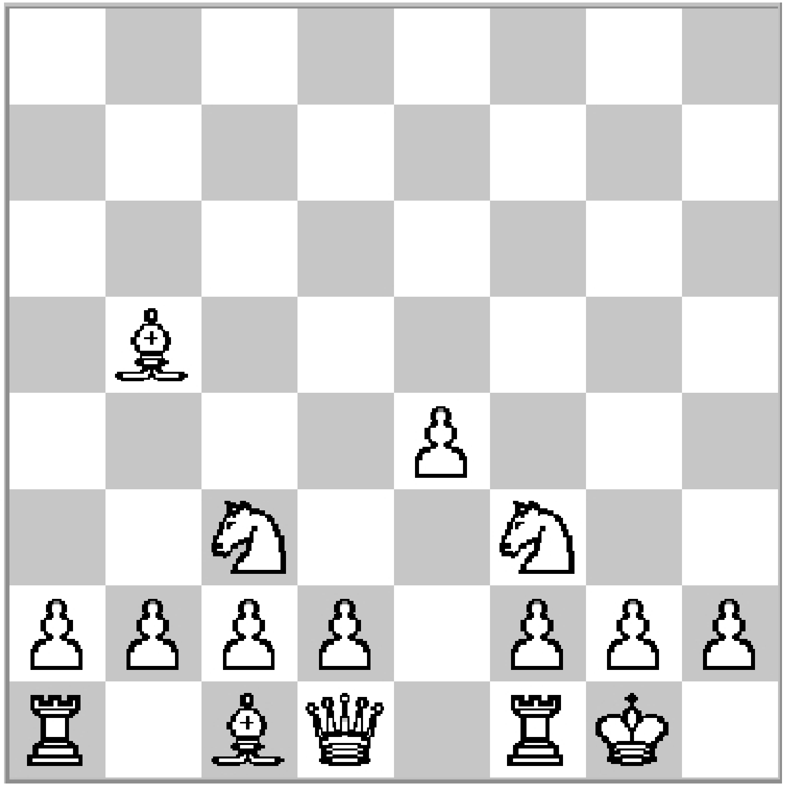
Backgammon

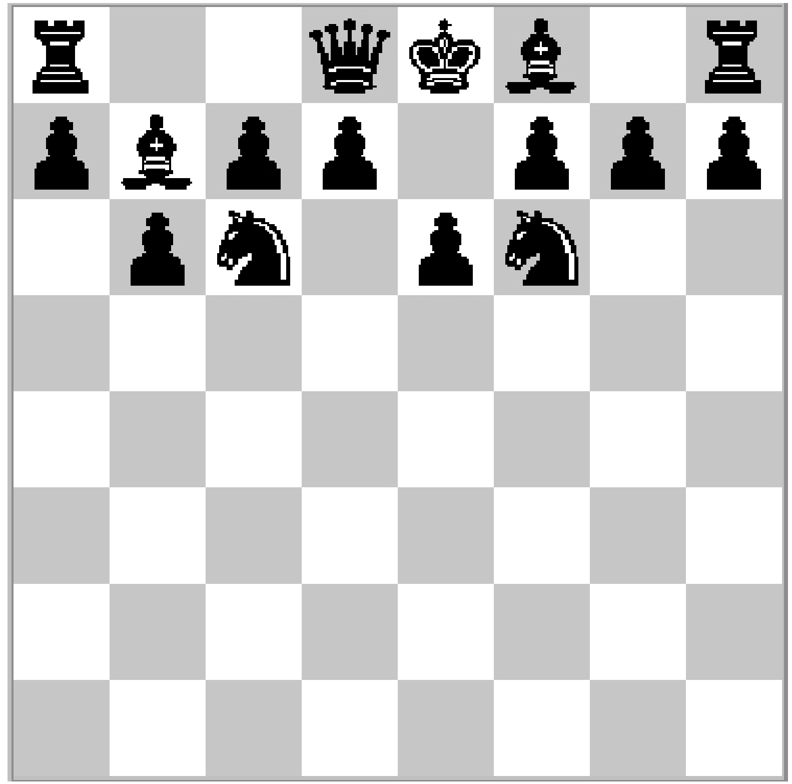


Maze World



Kriegspiel

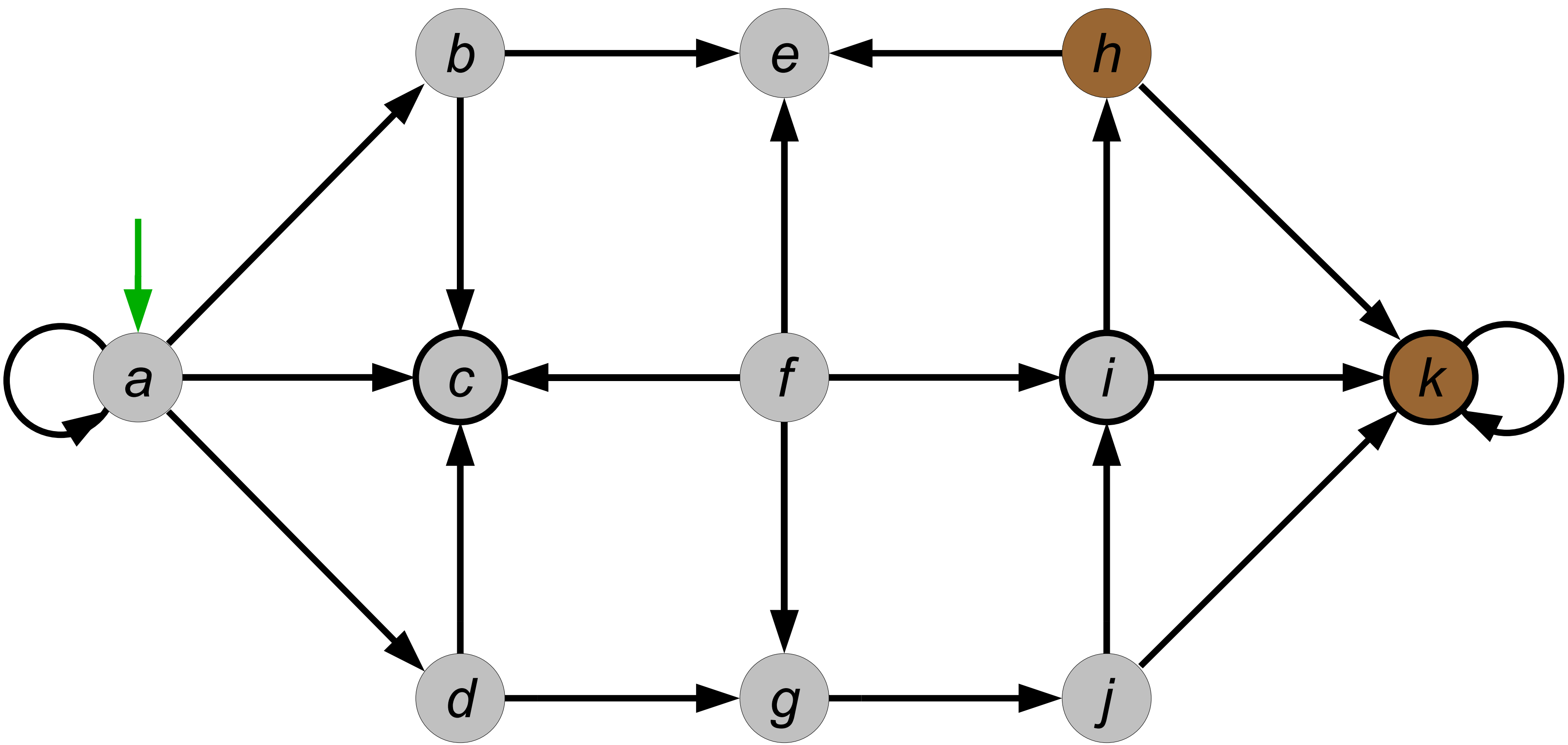




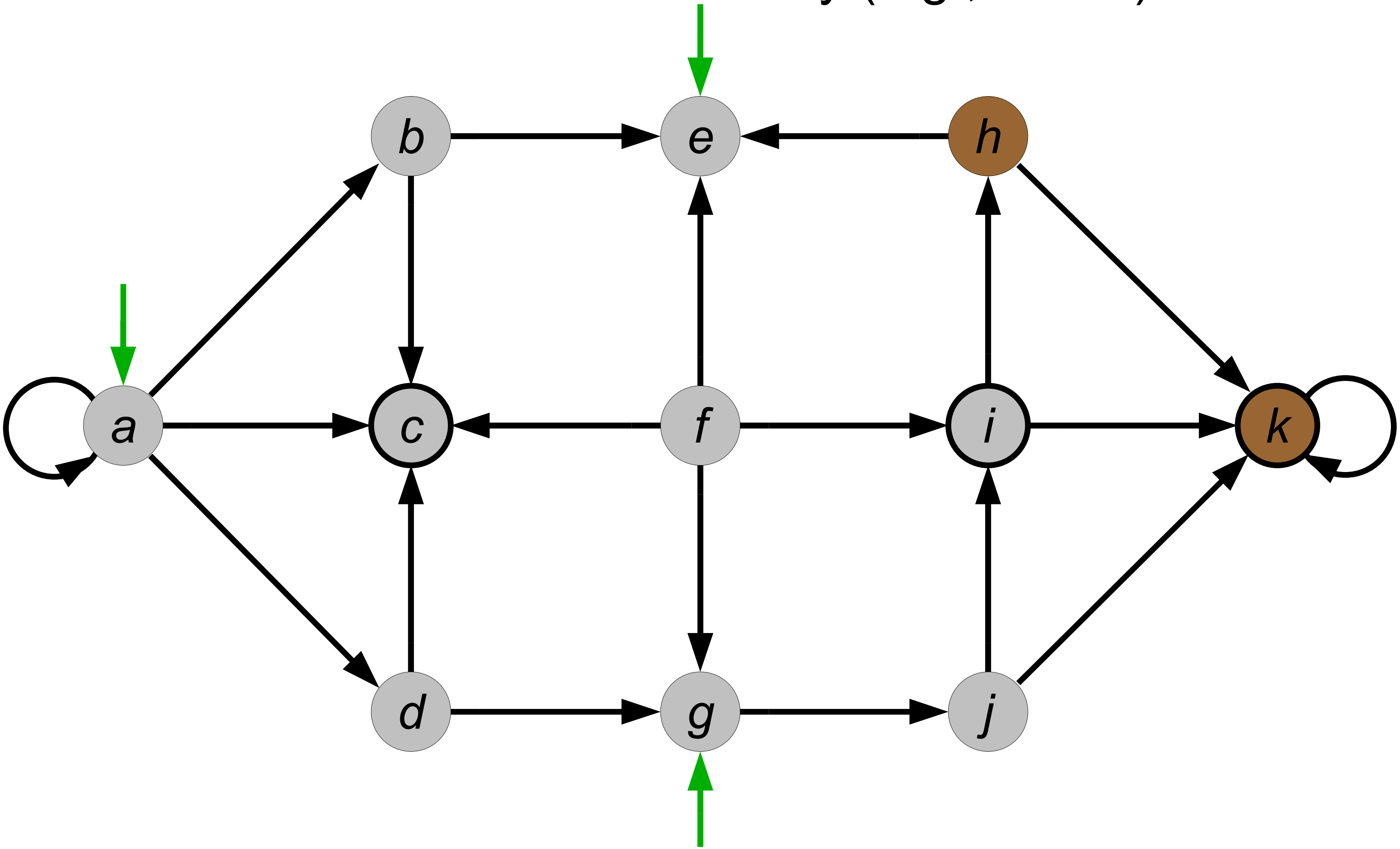
Poker


LindaS4
$5592.68
28
Current Hand #104673461
blacmax
$1000
rugge25
(sitting out)
stuar164
(sitting out)
PACIFIC POKER
www.888.com
lucster
$4787
DvlDog72
$107449.46
$417
Jockey21
$42956.55
Call
jelco1
$1222
andyffc
$2976

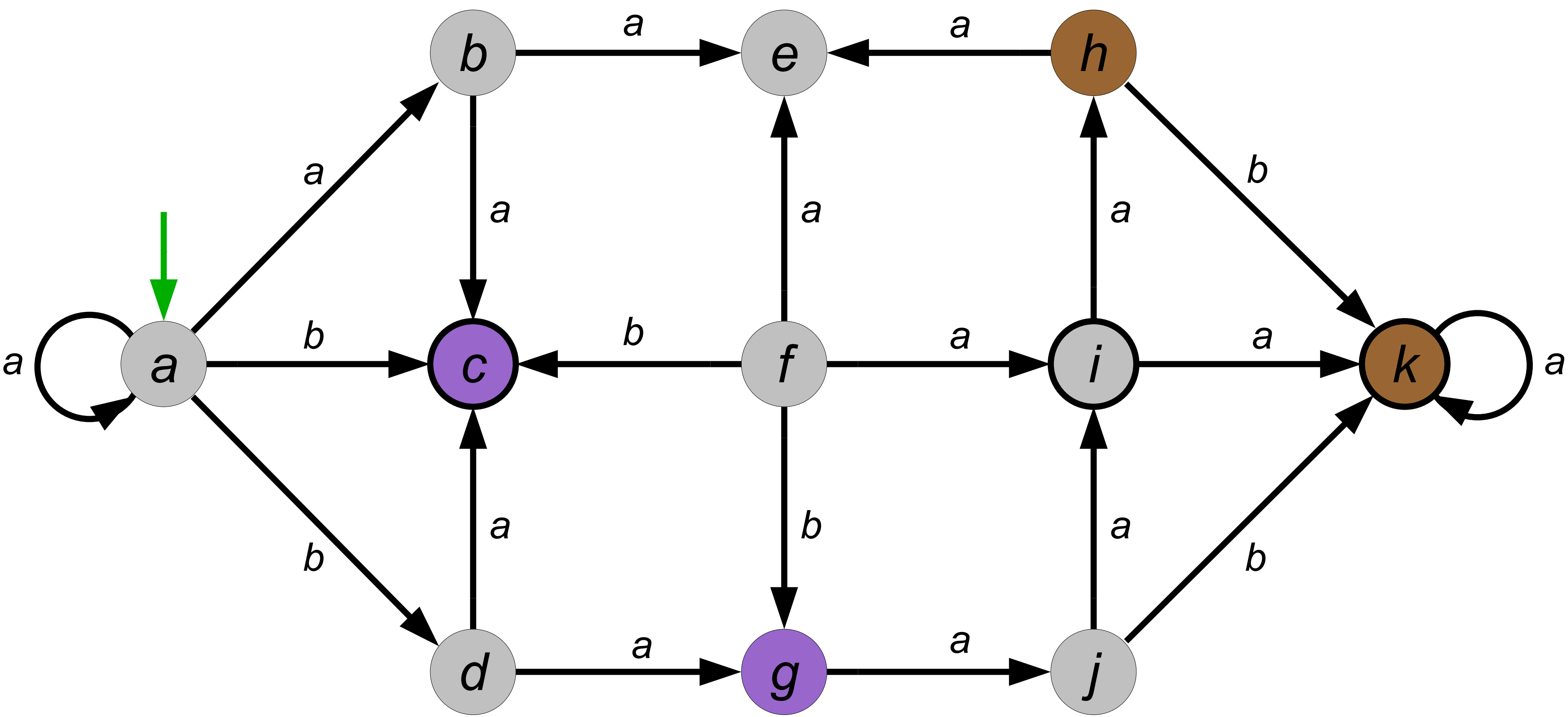
State Machine Model



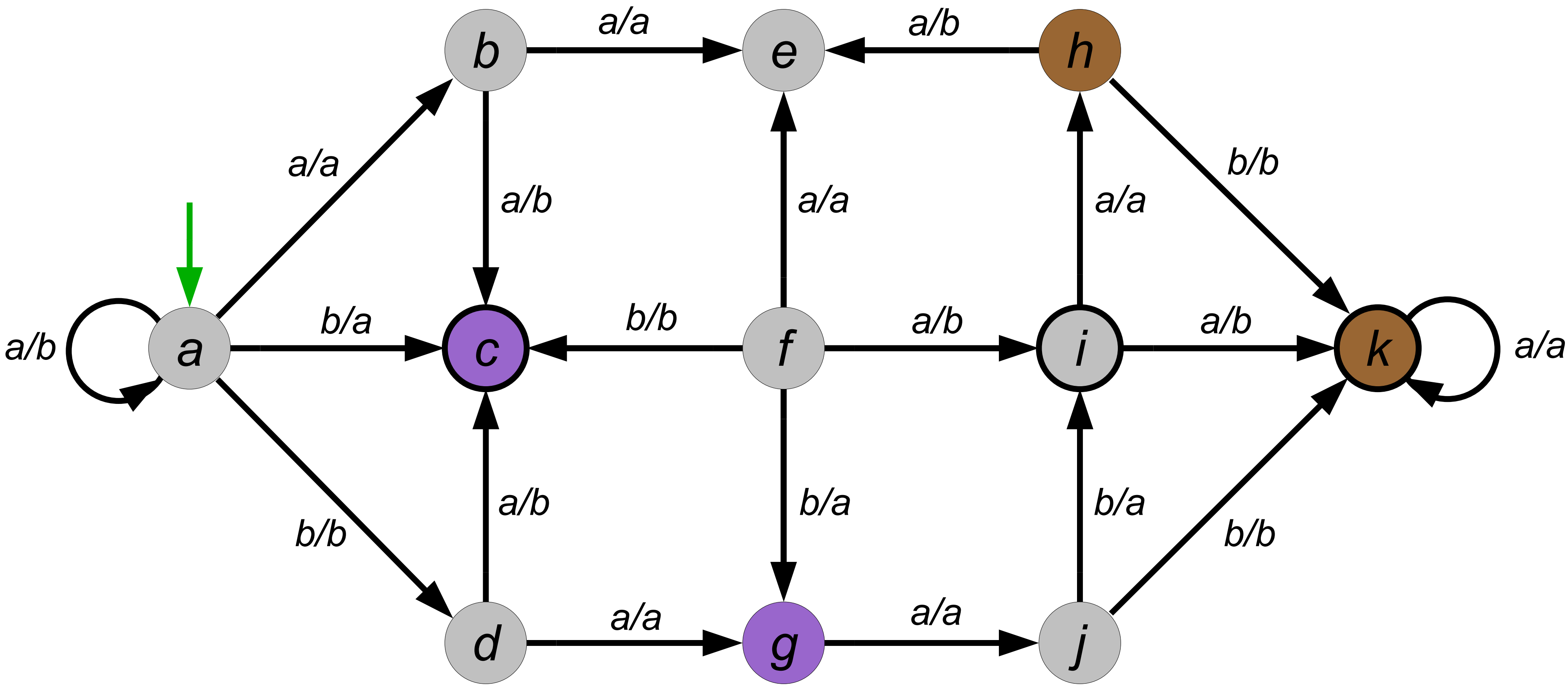
Initial State Uncertainty (e.g., Poker)



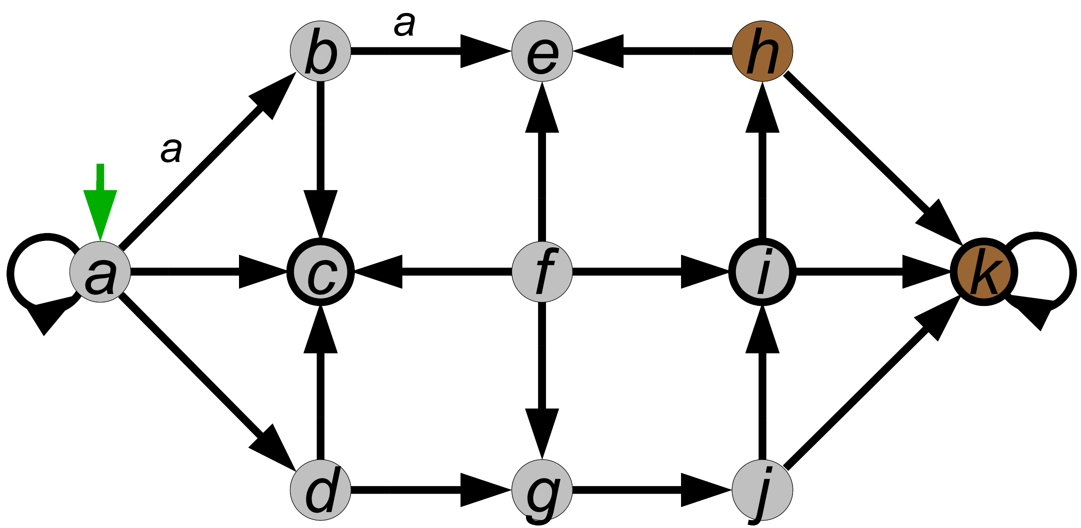
Nondeterministic Moves (e.g., Rolling a Dice)

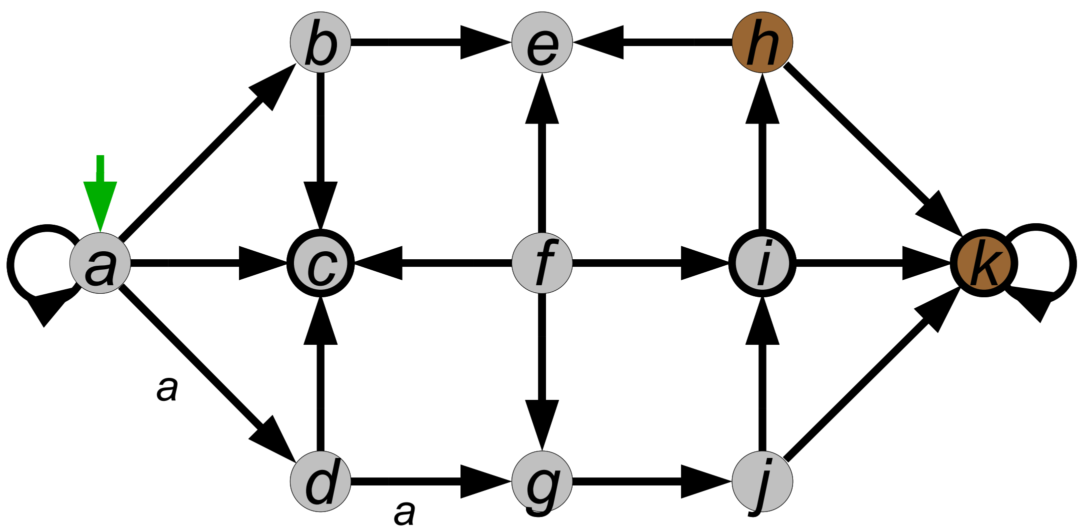


Action Uncertainty (any $n \geq 2$-Player Game)

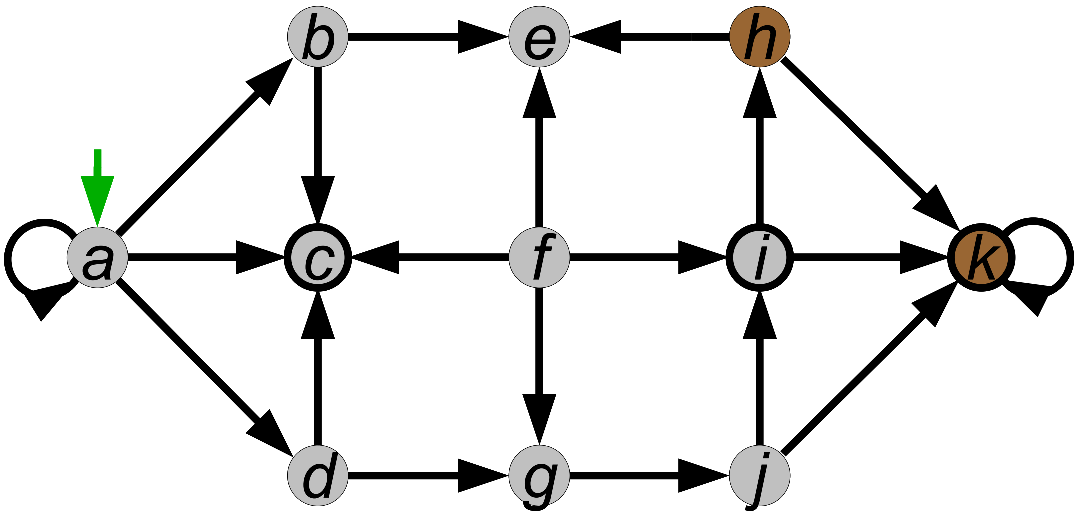


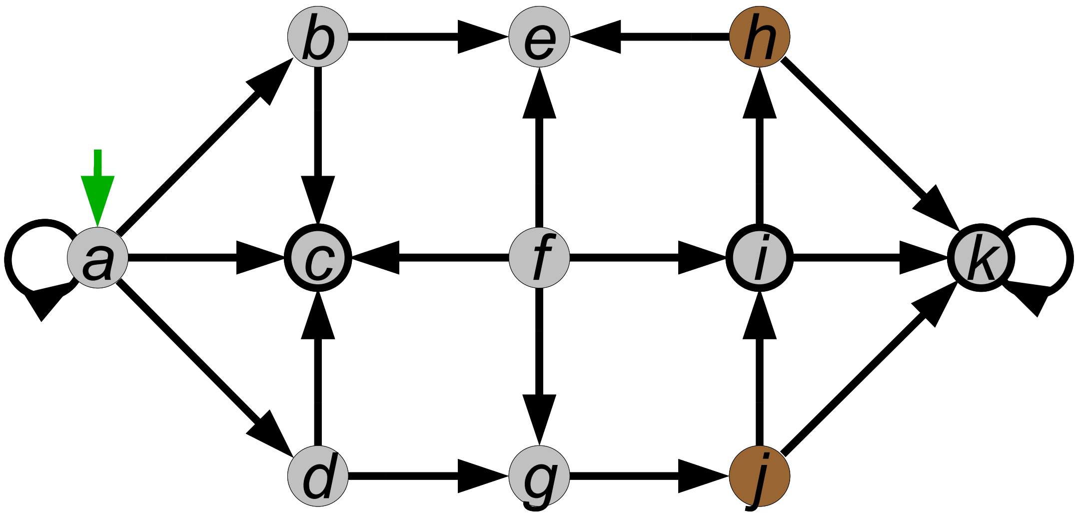
State Model Uncertainty





Termination and Goal Uncertainty





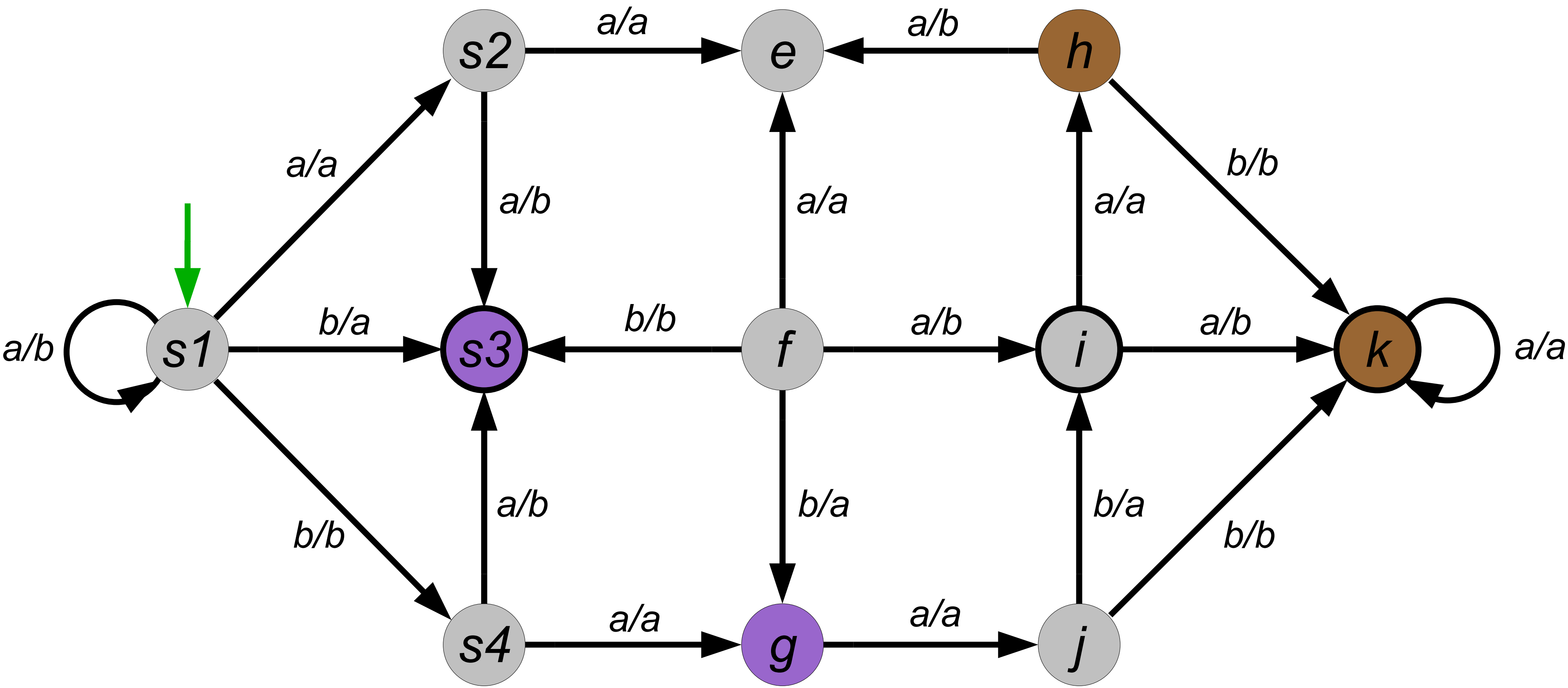
Problems with Incomplete Information

- Dealing with incomplete information can be costly, as multiple options must be considered.
- In the face of incomplete information, there may be no way of knowing that one has succeeded.

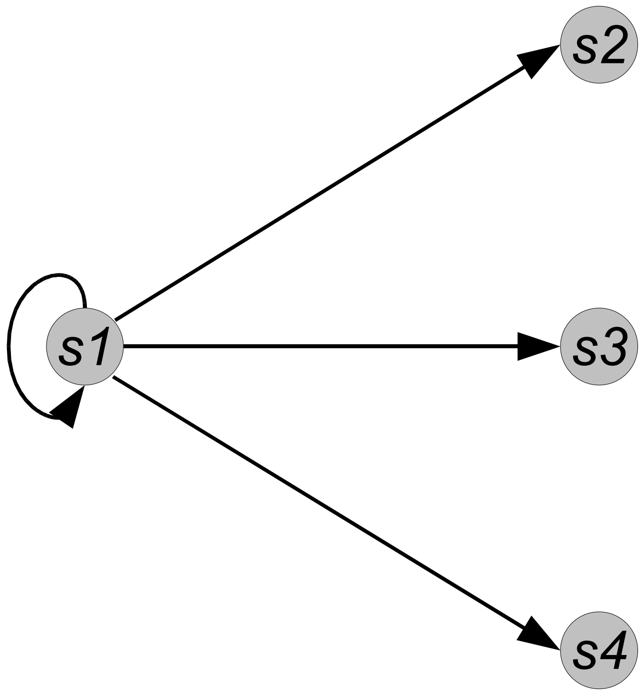
In GGP, it is customary to ensure that the players have enough information to determine legality of moves, termination, and goals.

(Note: Kriegspiel requires an arbiter to ensure legal play.)

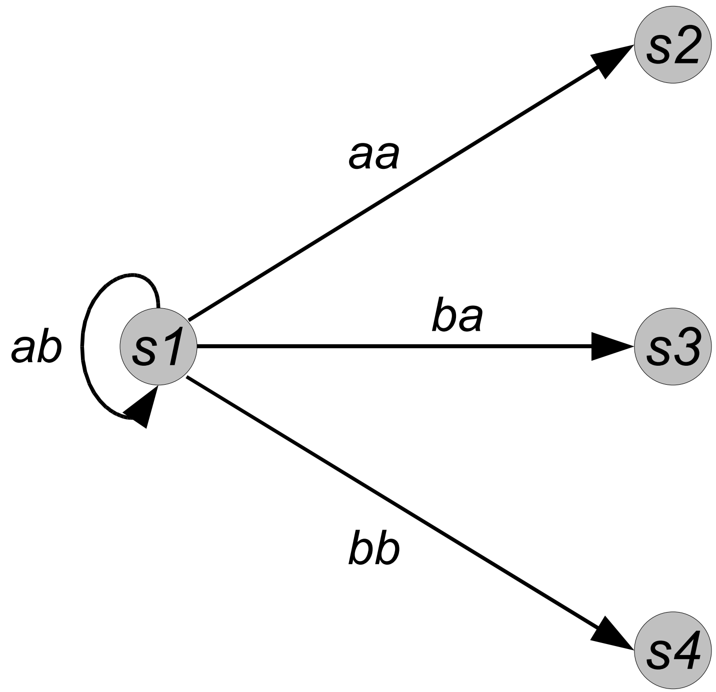
State-Space Search with Multiple Players



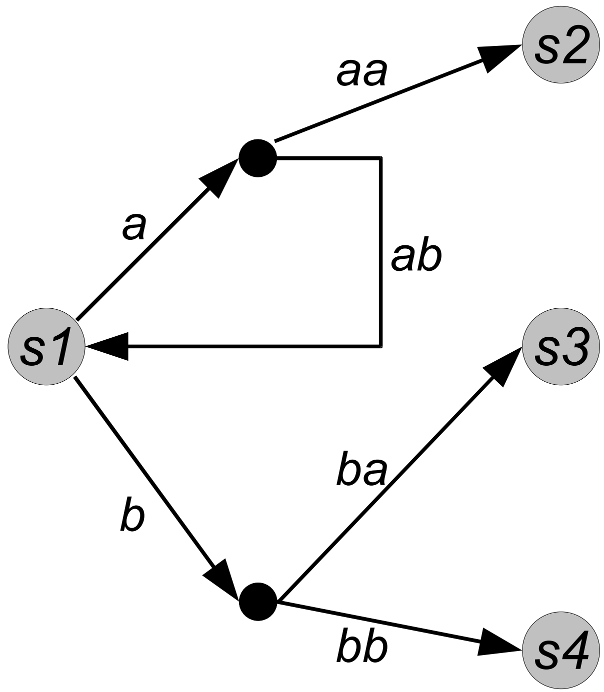
Single Player Game Graph



Multiple Player Game Graph



Bipartite Game Graph



Move Lists

Simple move list

```
[(a,s2),(b,s3)]
```

Multiple player move list

```
[([a,a],s2),([a,b],s1),
 ([b,a],s3),([b,b],s4)]
```

Bipartite move list

```
[(a,[([a,a],s2),([a,b],s1)]),
 (b,[([b,a],s3),([b,b],s4)])]
```

Single Player Node Expansion (from Previous Lecture)

```
function expand (node)
var match,role,old,data,al,nl,a
begin
    match := node.match; role := match.role; al := []; nl := [];
    for a in legals(role,node) do
        data := simulate(node,{does(role,a)});
        new := create_node(match,data,node.theory,node,[],-1);
        if terminal(new) then new.score := goal(role,new);
        nl := {new} ∪ nl;
        al := {(a,new)} ∪ al
    end-for;
    node.alist := al; return nl
end
```

Multiple Player Node Expansion

```
function expand (node)
var match,role,old,data,al,jl,nl,a
begin
    match := node.match; role := match.role; al := []; jl := []; nl := [];
    for a in legals(role,node) do
        for j in joints(role,a,match.roles,node) do
            data := simulate(node,jointactions(match.roles,j));
            new := create_node(match,data,node.theory,node,[],-1);
            if terminal(new) then new.score := goal(role,new);
            nl := {new} ∪ nl;
            jl := {(j,new)} ∪ jl
        end-for;
        al := {(a,jl)} ∪ al
    end-for;
    return nl
end
```

New Subroutines

```
function joints (role,action,roles,node)   % returns the combinatorial list of all joint actions
var jl := [ ]                              % where role does action
begin
    if roles =[ ] then return [ [ ] ];
    for al in joints(role,action,tail(roles),node) do
        if head(roles) = role then jl := {{action} ∪ al} ∪ jl
        else for x in legals(head(roles),node) do
            jl := {{x} ∪ al} ∪ jl;
    return jl
end

function jointactions(roles,j)             % returns set of does atoms
begin                                      % for joint action j
    if roles = [ ] then return [ ]
    else return {does(head(roles),head(j))} ∪ jointactions(tail(roles),tail(j))
end
```

Example

Call: *joints*`(1,a,[1,2],s1)`
    Call: joints`(1,a,[2],s1)`
        Call: *joints*`(1,a,[],s1)`
        Exit: `[[]]`
    Exit: `[[a],[b]]`
Exit: `[[a,a],[a,b]]`

Call: *jointactions*`([1,2],[a,b])`
    Call: *jointactions*`([2],[b])`
        Call: *jointactions*`([],[])`
        Exit: `[]`
    Exit: `[does(2,b)]`
Exit: `[does(1,a),does(2,b)]`

Best Move (Multiple Player Games)

```
function bestmove (node)
var max,score,best,a,jl
begin
    max := 0;
    (best,jl) := head(node.alist);
    for (a,jl) in node.alist do
        score := minscore(jl);
        if score = 100 then return a;
        if score > max then
            max := score; best := a
        end-if
    end-for;
    return best
end
```

Note: This makes the pessimistic assumption that the other players make the most harmful (for us) joint move.

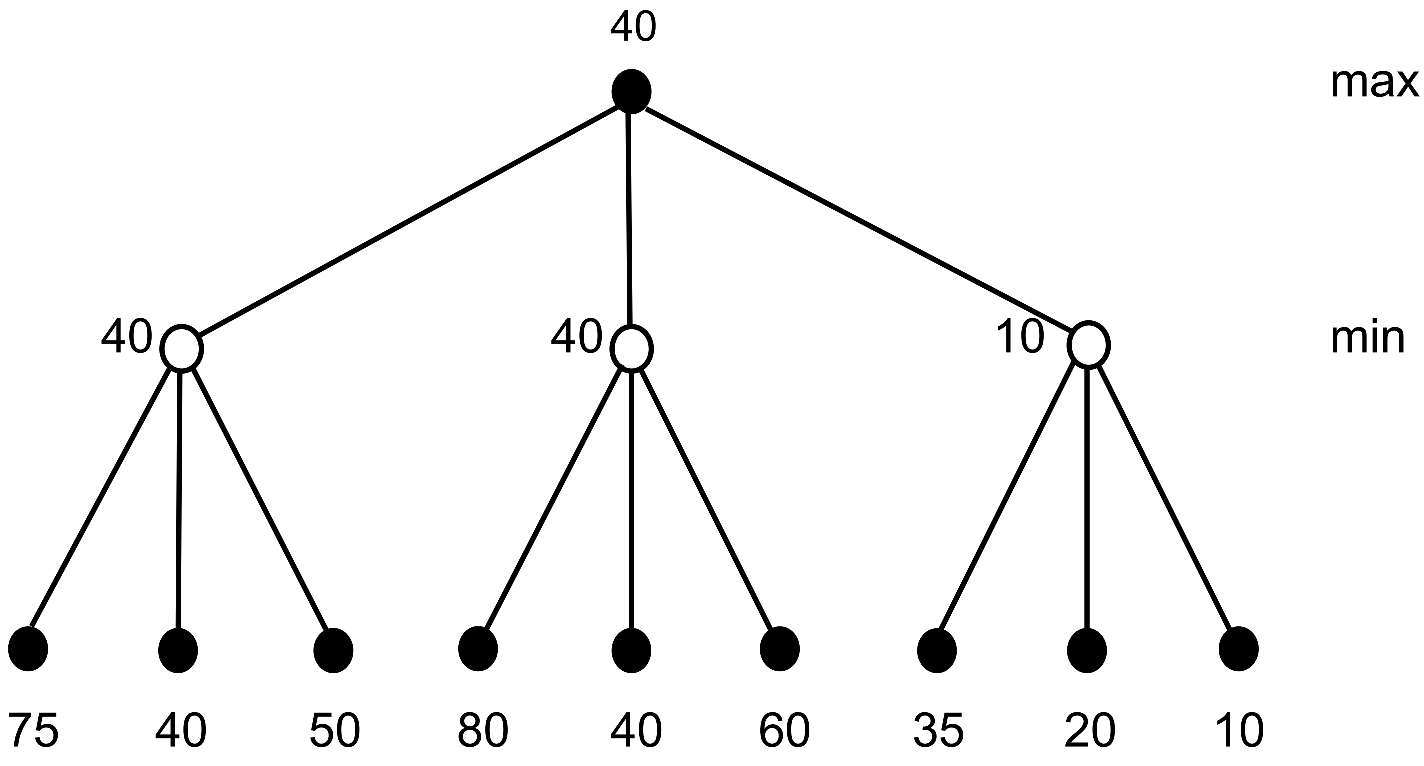
Node Evaluation (Joint Moves)

```
function minscore (jl)
var min,score,j,child
begin
    min := 100;
    for (j,child) in jl do
        score := maxscore(child);
        if score = 0 or score = -1 then return score;
        if score < min then min := score
    end-for;
    return min
end
```

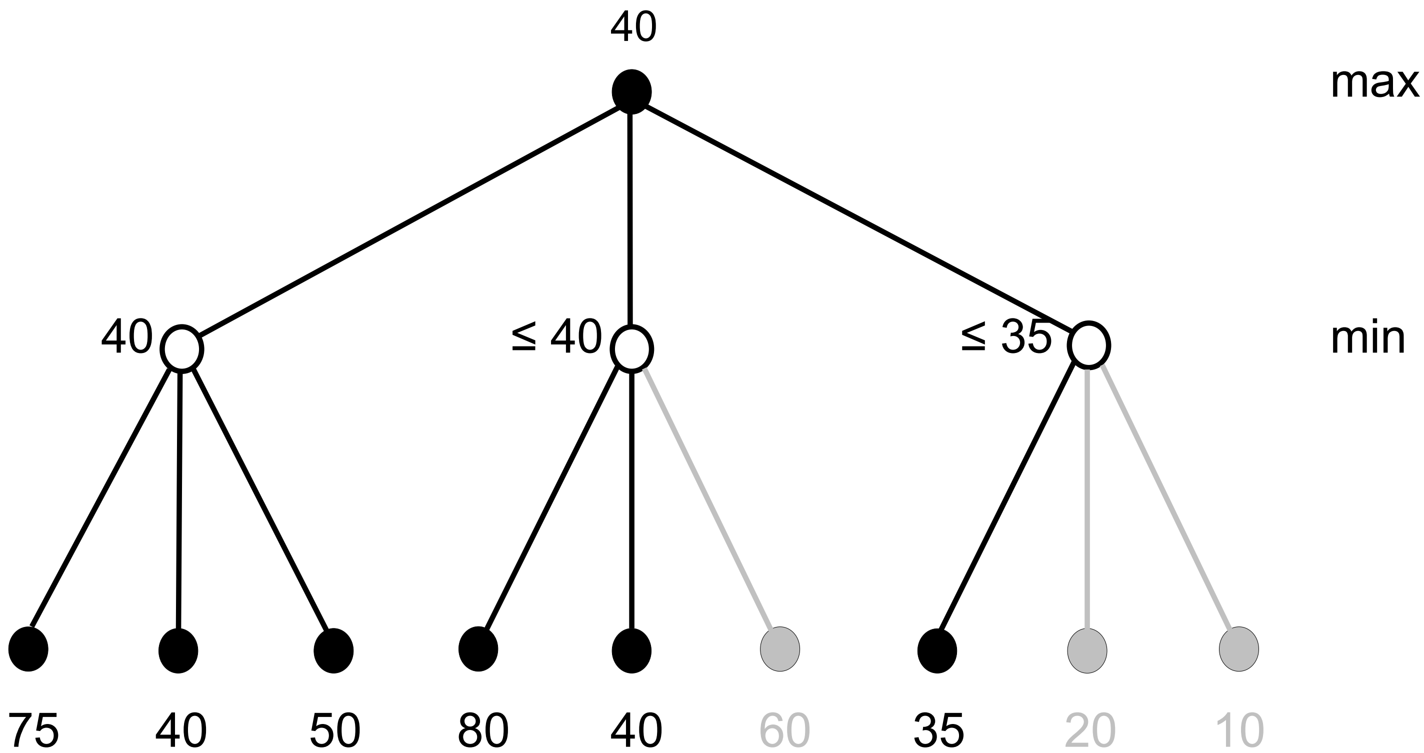
Node Evaluation (Our Moves)

```
function maxscore (node)
var max,score,a,jl,child
begin
    if node.score > -1 then return node.score;
    if node.alist = [ ] then return -1;
    max := 0;
    for (a,jl) in node.alist do
        score := minscore(jl);
        if score = 100 or score = -1 then return score;
        if score > max then max := score
    end-for;
    return max
end
```

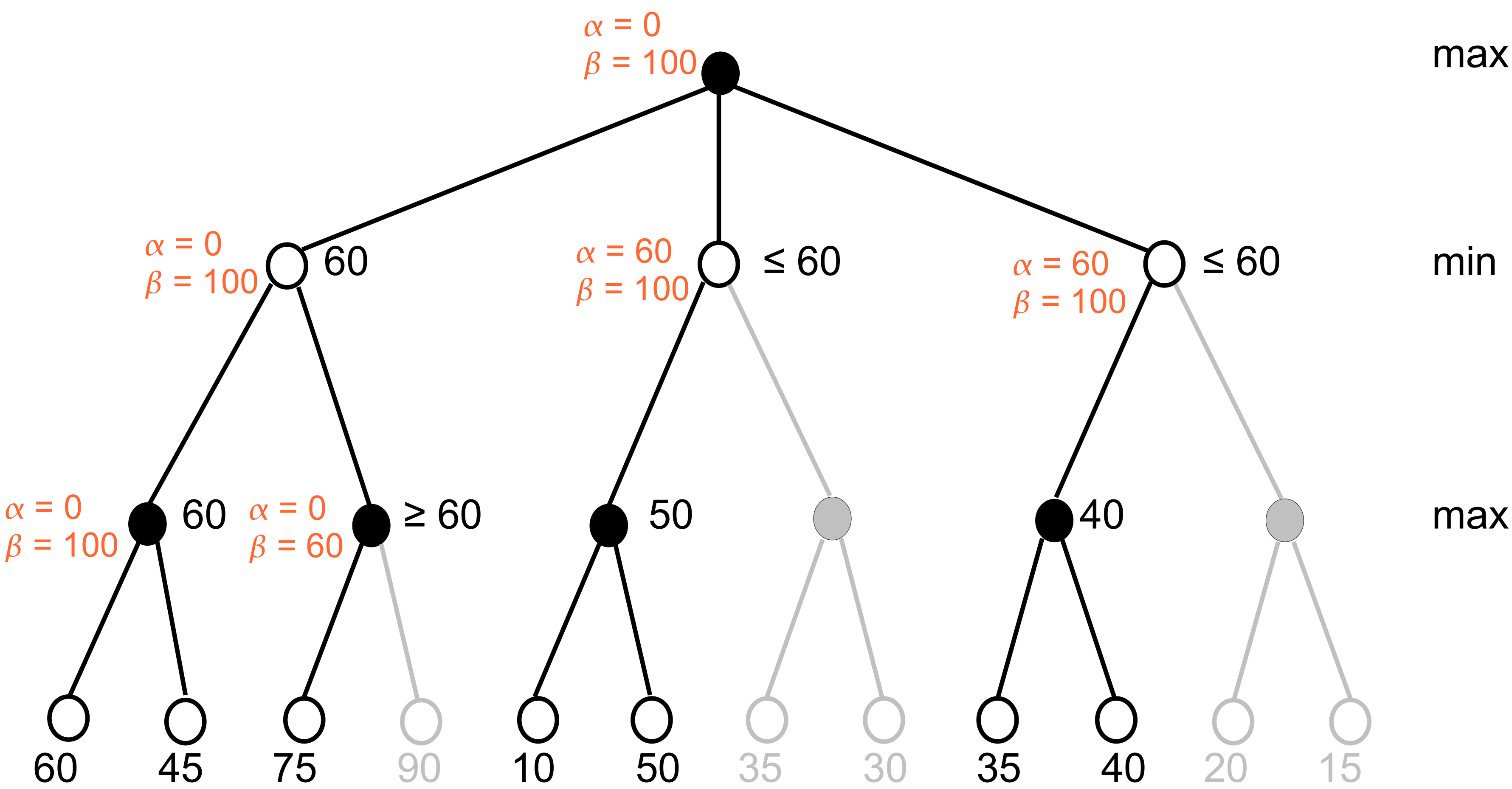
Minimax for Two-Person Zero-Sum Games



The α-β-Principle: α-Cutoffs



The α-β-Principle: α- and β-Cutoffs



Non Zero-Sum Games

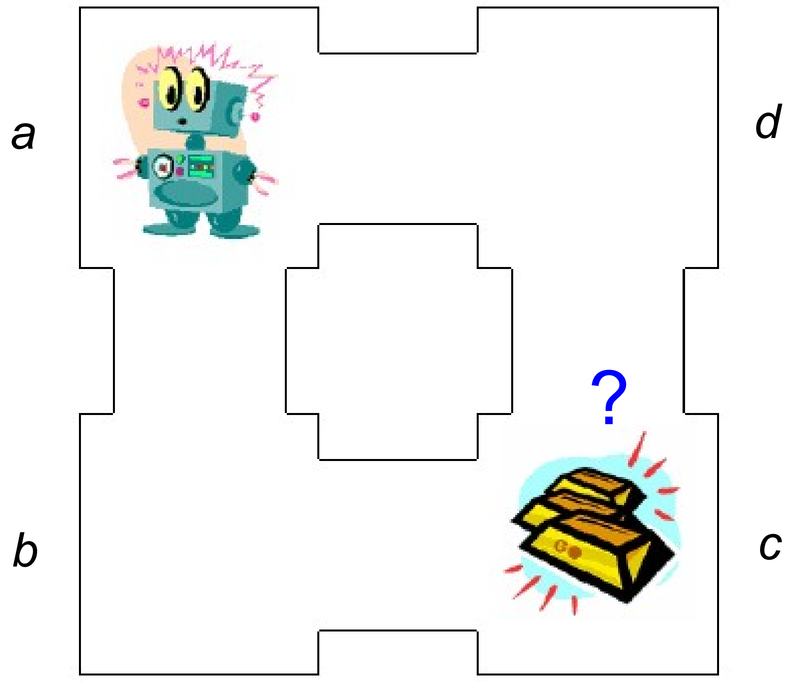
Problem:

All these techniques assume that the other players together choose the joint move that is most harmful for us. This is often too pessimistic a model for the opponents in other than two-person zero-sum games.

Solution:

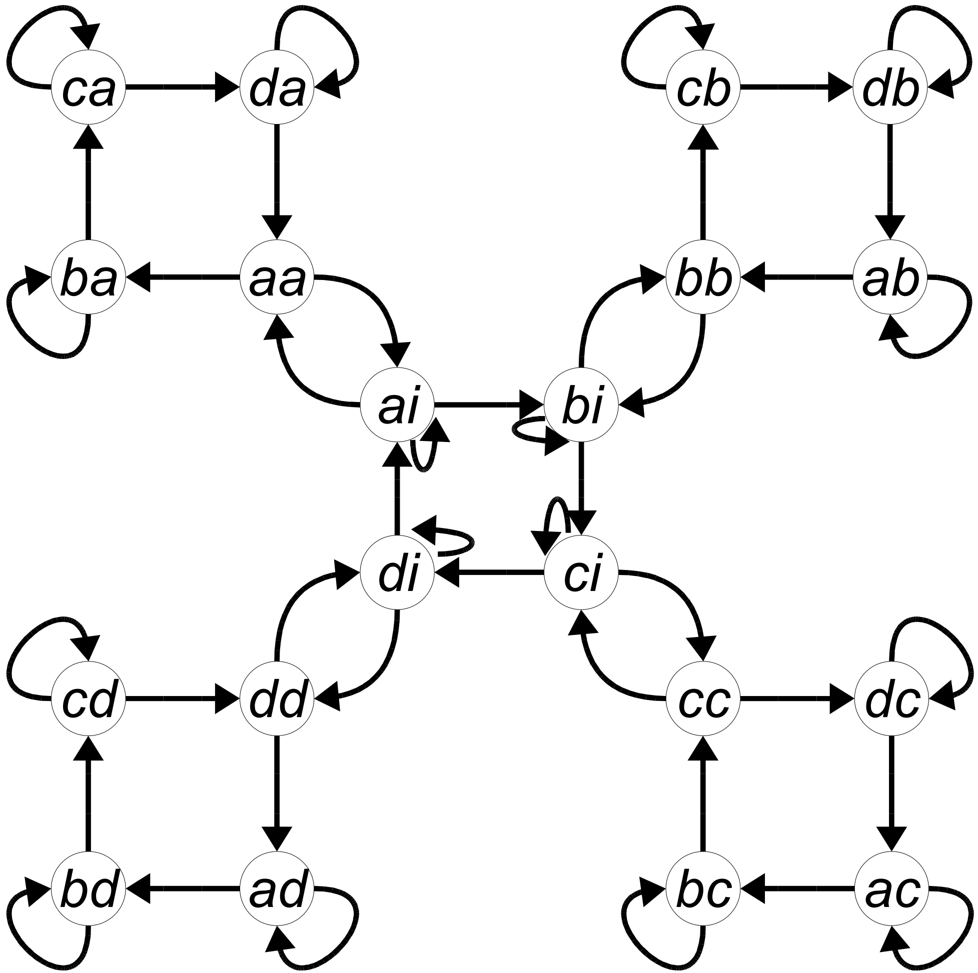
Game Theory (Lecture 7)

Maze World

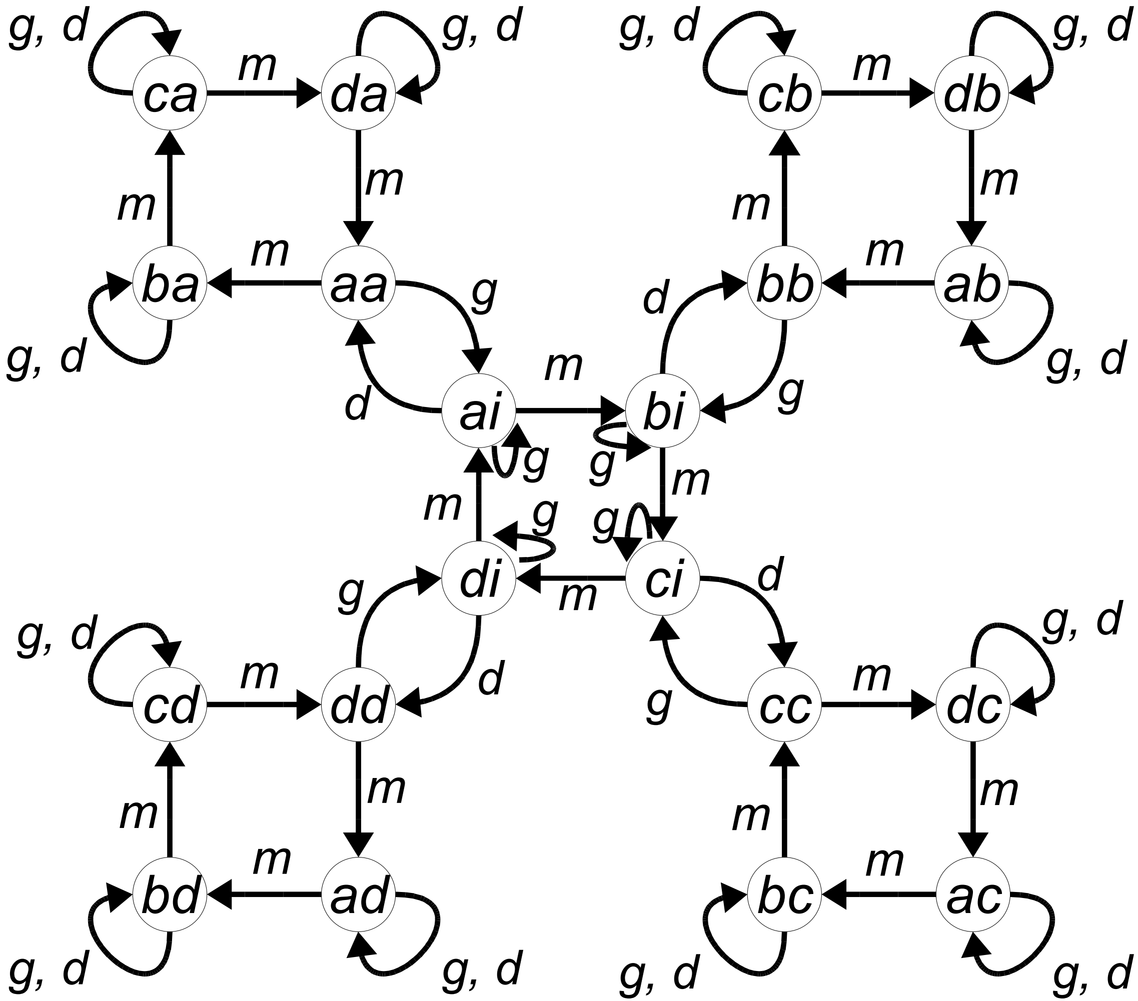


Initial State: *ac* (robot in *a*, gold in *c*)

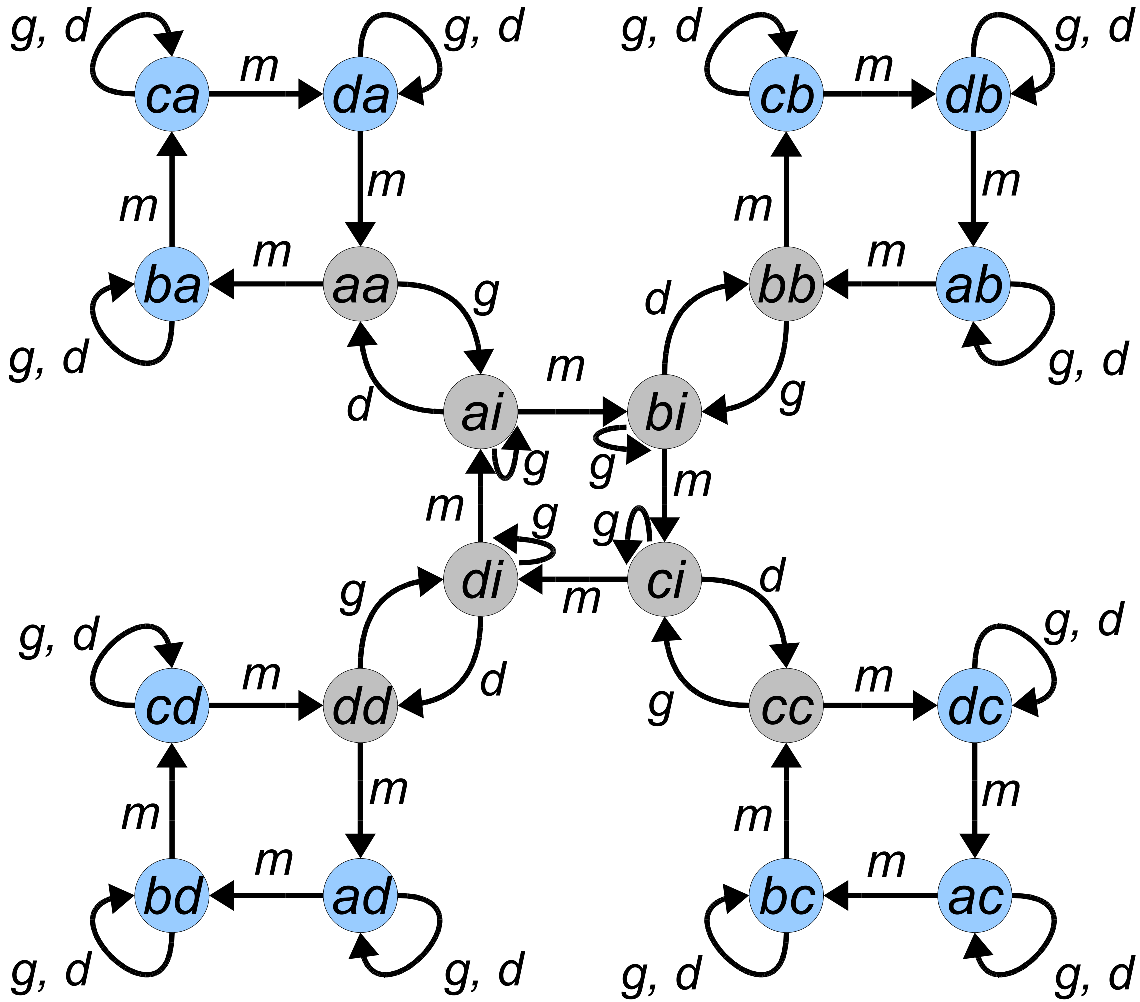
Environment Model



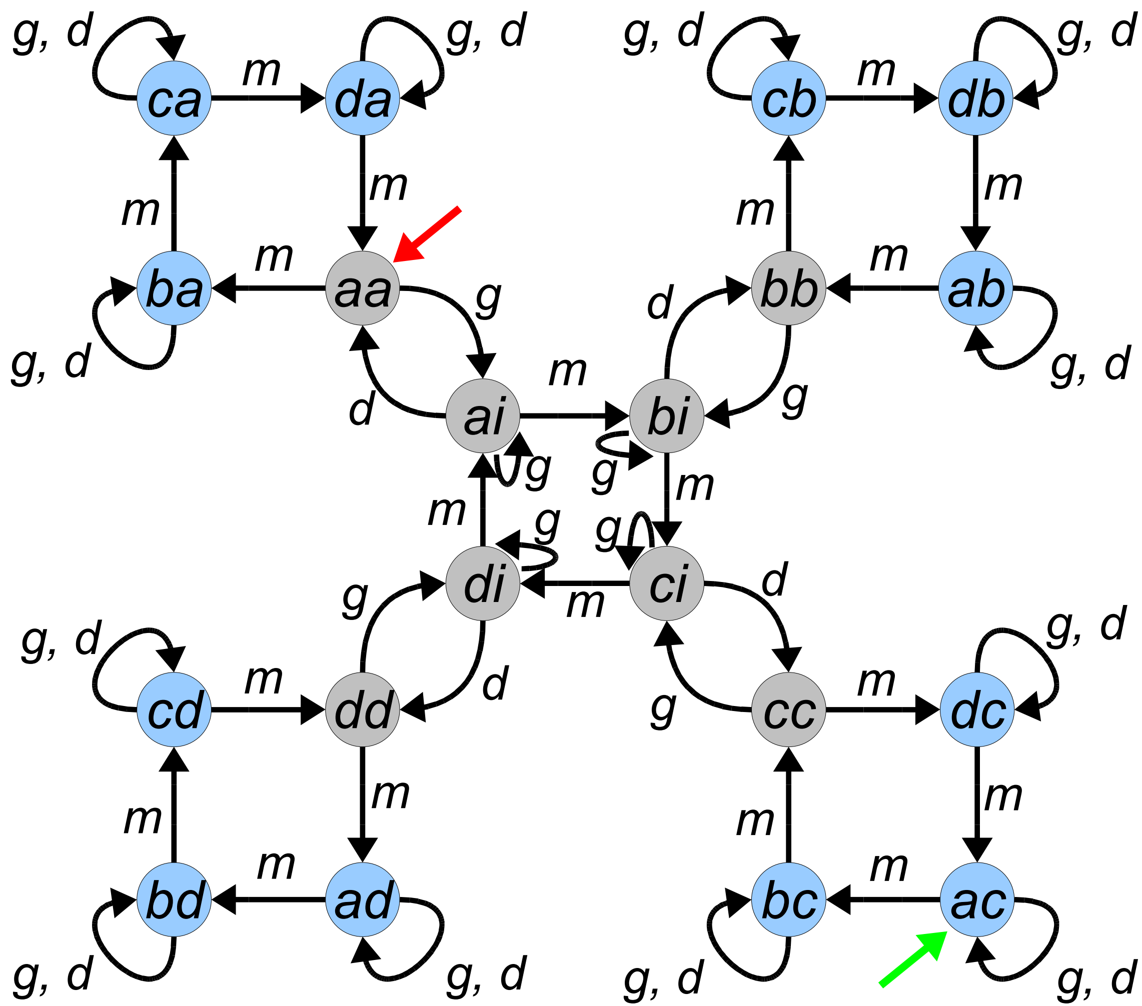
Agent Actions



Agent Percepts

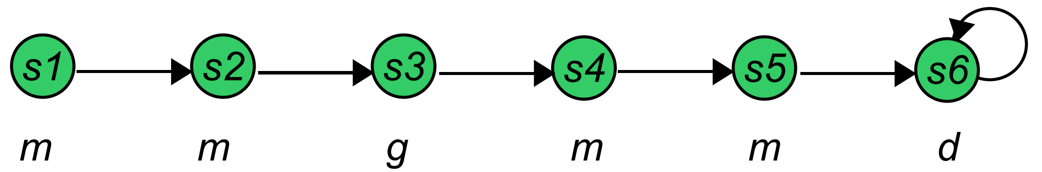


Initial States and Goals



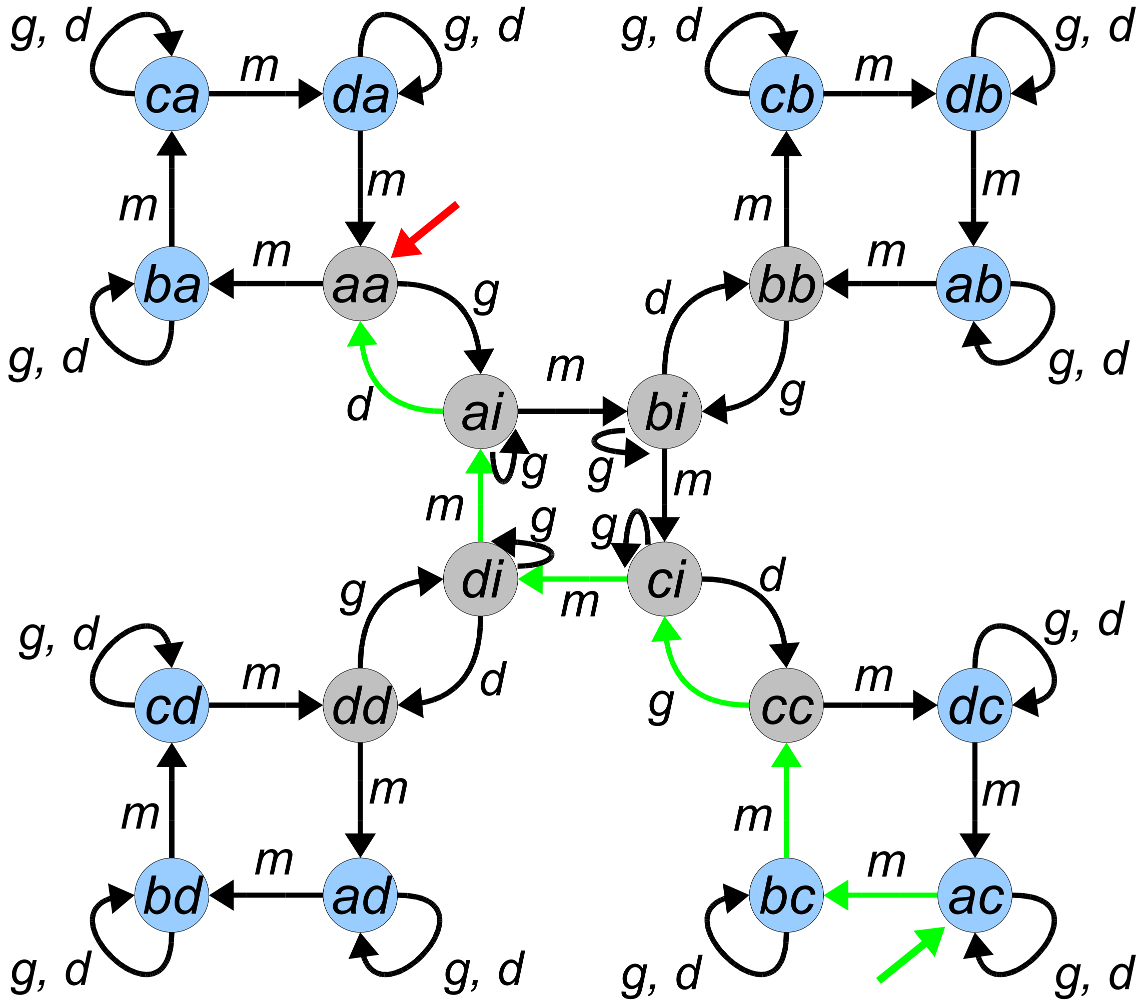
Planning

Planning is the process of finding a transition diagram *for our agent* that causes its environment to go from any initial state to a goal state.



Planning can be done *offline* and the resulting plan/ program installed in the agent *or* the planning can be done *online* followed by execution.

State Space Planning



Incompleteness

Incompleteness

- Initial state
- Transition diagram for environment
- Goal

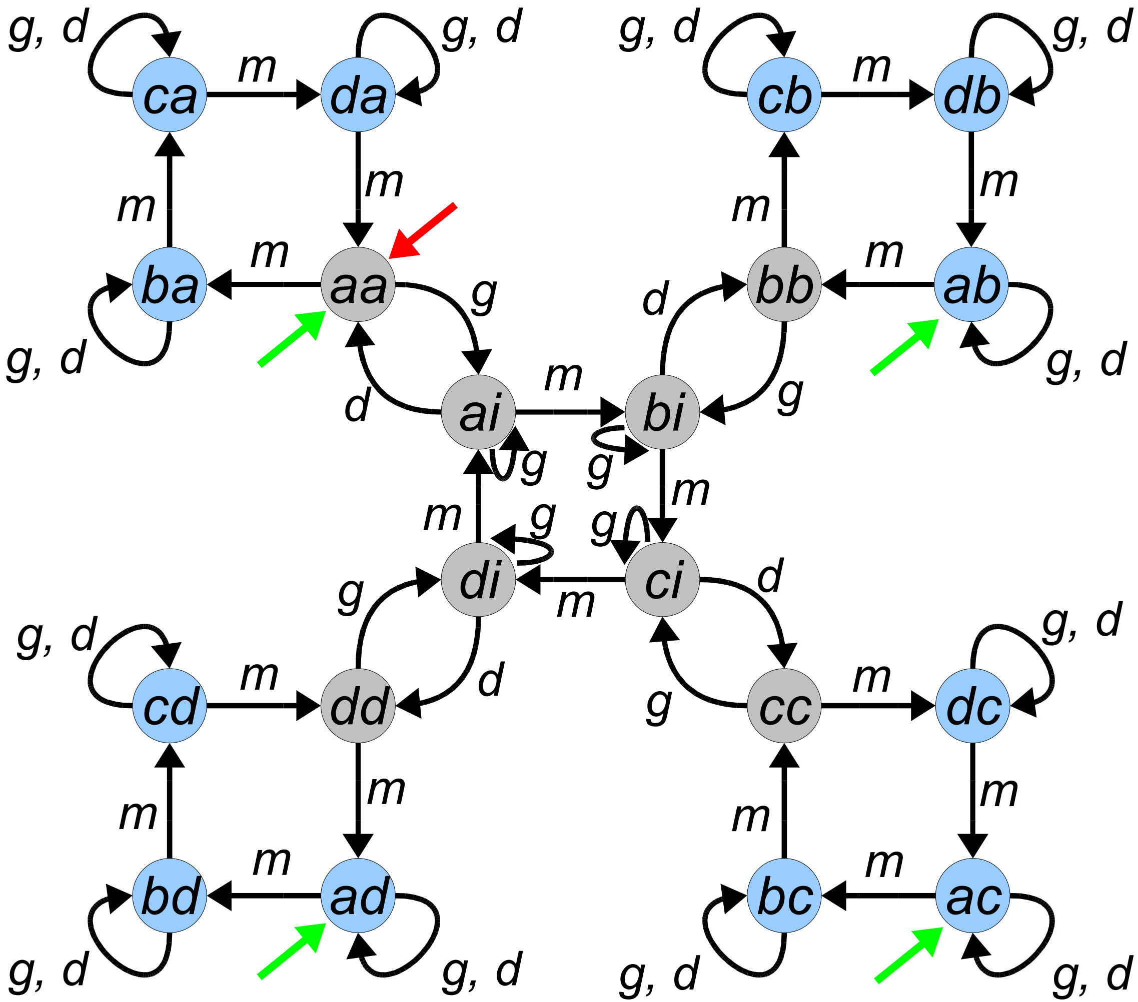
Complete Planning Techniques

- Coercion (e.g. do the *grab* move at all locations)
- Conditional plan (e.g. if see the gold grab it; else go)

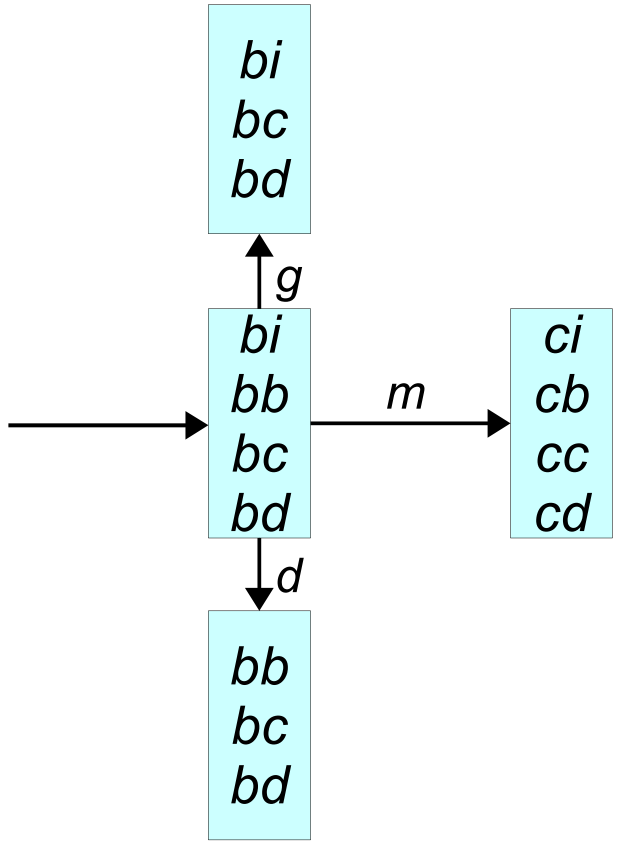
Postponement Techniques

- Delayed planning
- Assumption

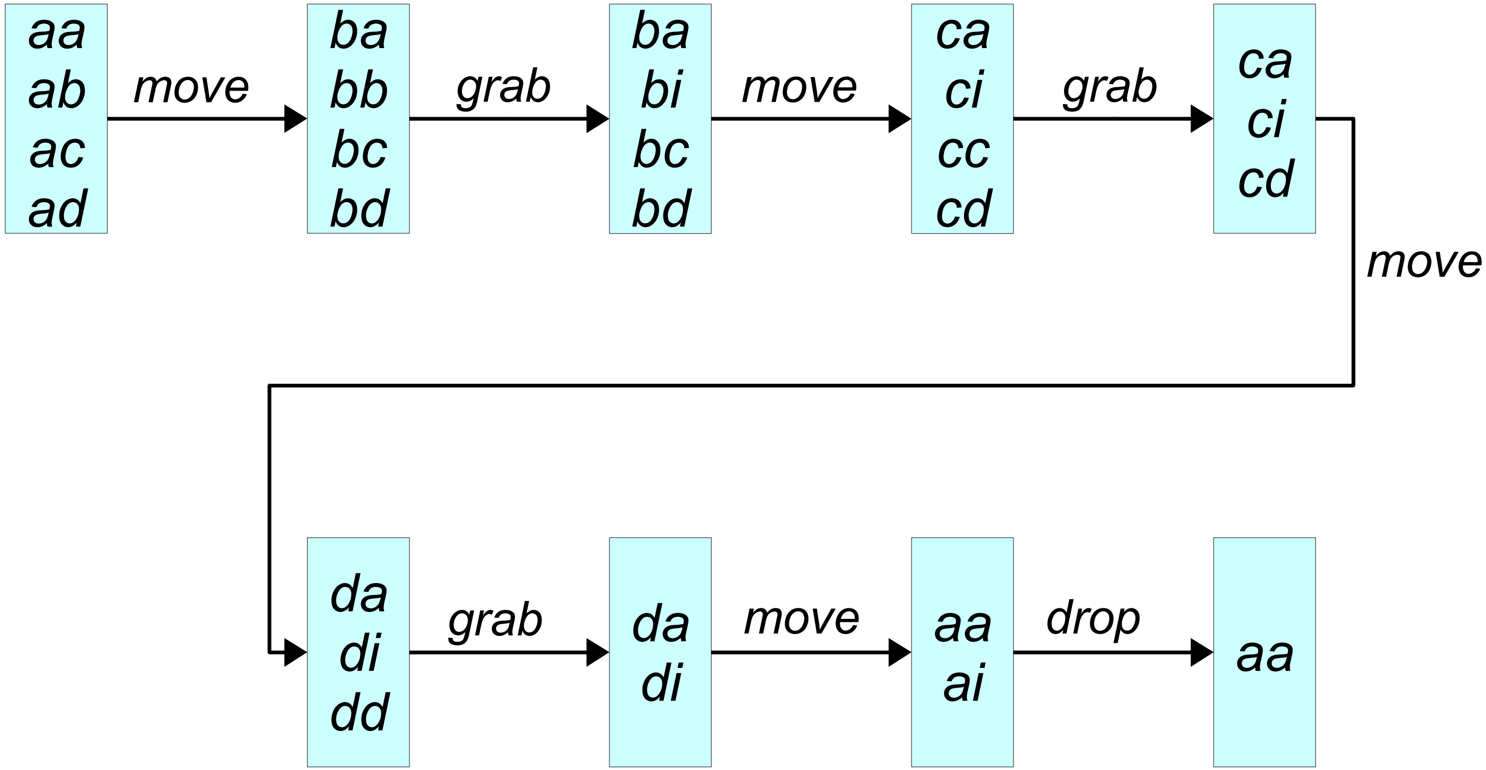
Initial State Uncertainty



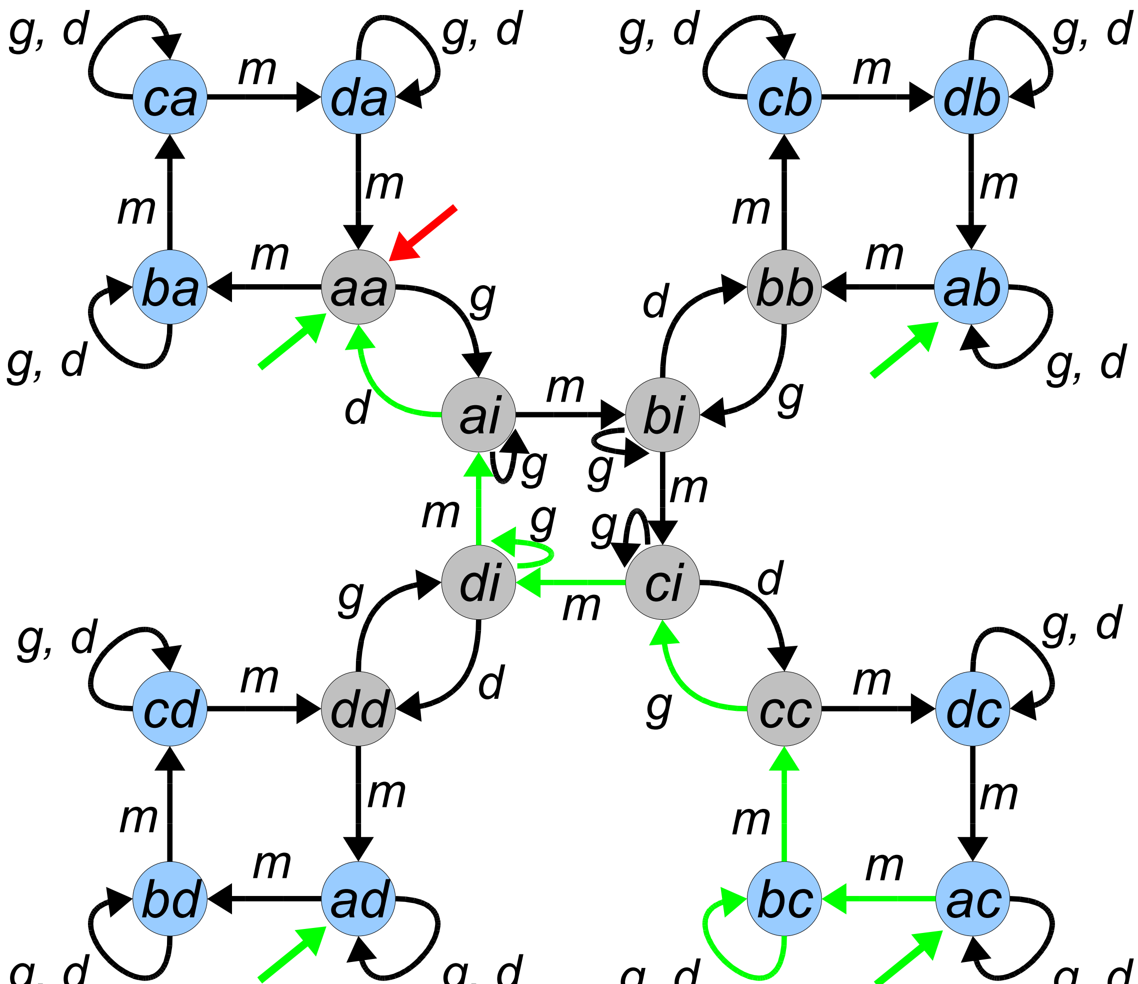
Sequential State Set Progression



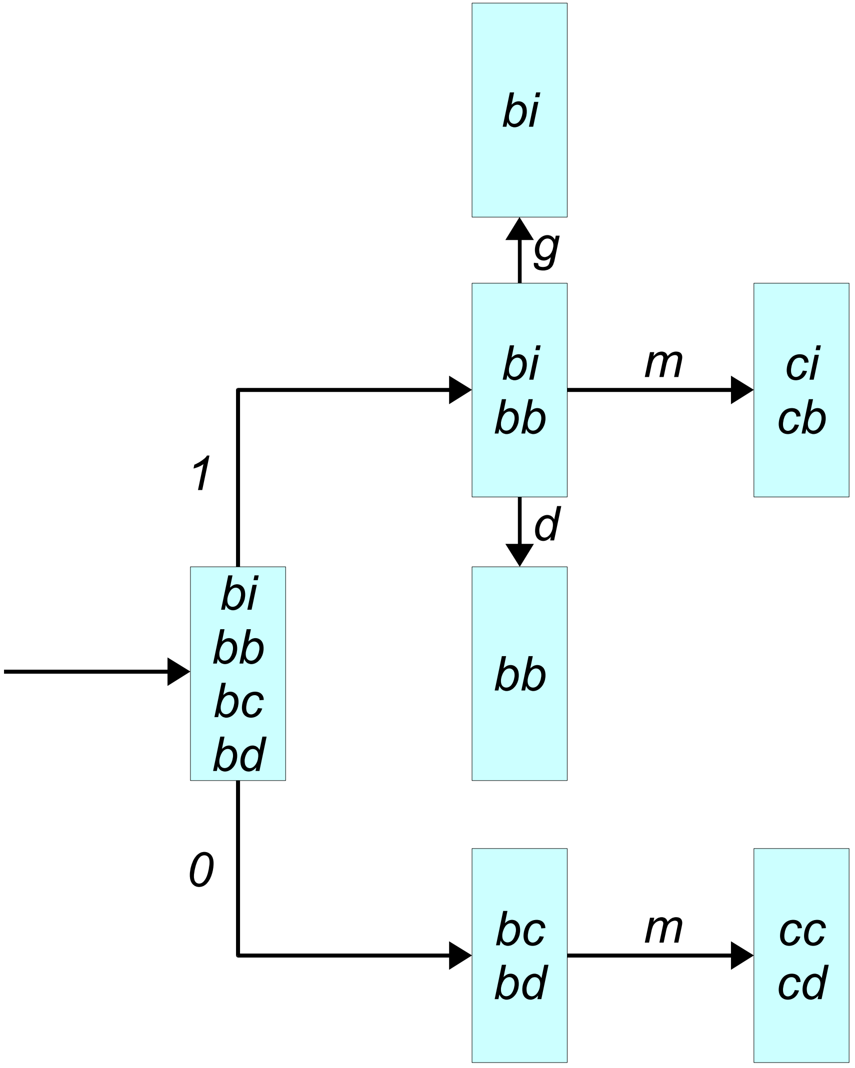
Sequential State Set Plan



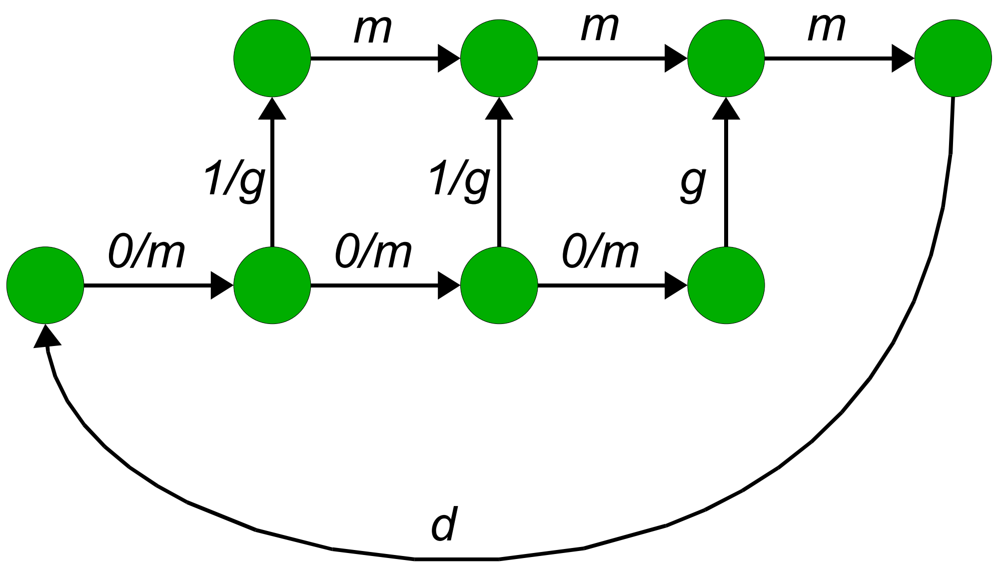
Execution



Conditional State Set Progression



Conditional State Set Plan



Comparison

Sequential plan

- possible that no plan exists
- plan may contain redundant moves

Conditional plan

- large search space

Delayed planning

- irreversibility problematic

Moral of this Story

As we can see from this analysis, it is sometimes desirable for an agent to do only a portion of its planning up front, secure in the knowledge that it can do more later as necessary.

Planning can be done *offline* and the resulting plan/program executed during play *or* the planning can be done *online* and interleaved with execution.