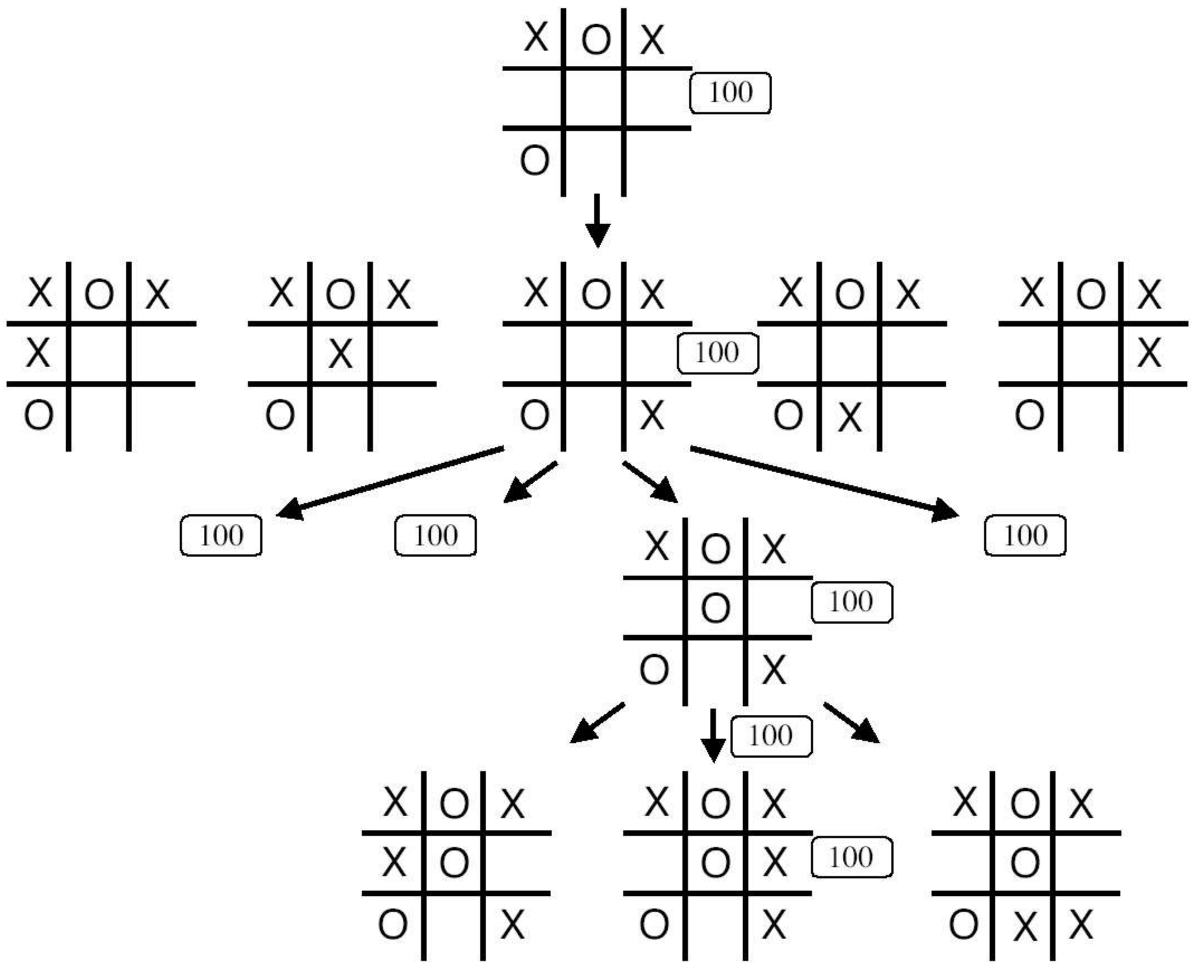
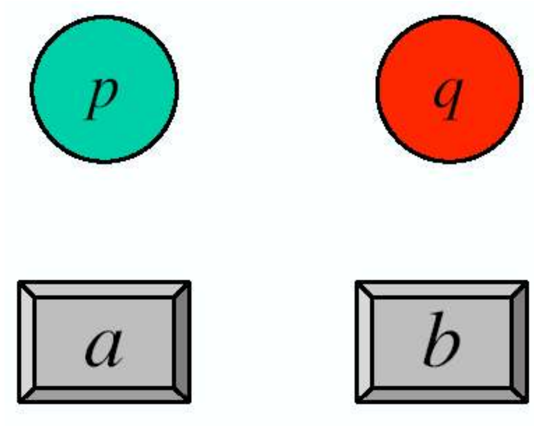
Lecture 3:
State-Space Search for Single Player Games

- Single Player Games with Complete Information
- Iterative Deepening
- Data Structures and a Generic Search Procedure

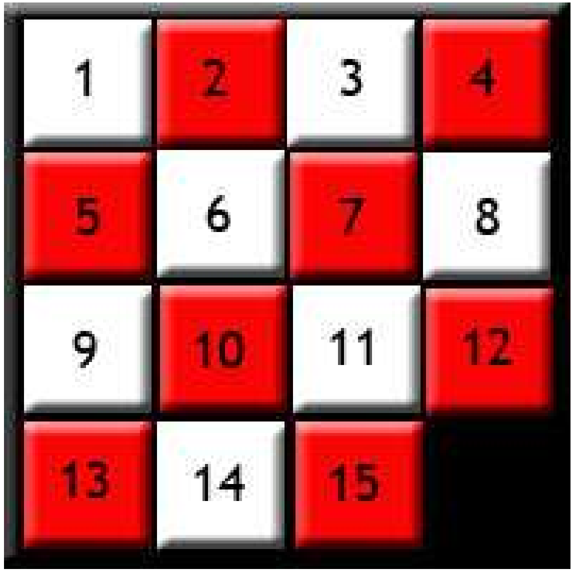
Game Tree Search (General Concept)



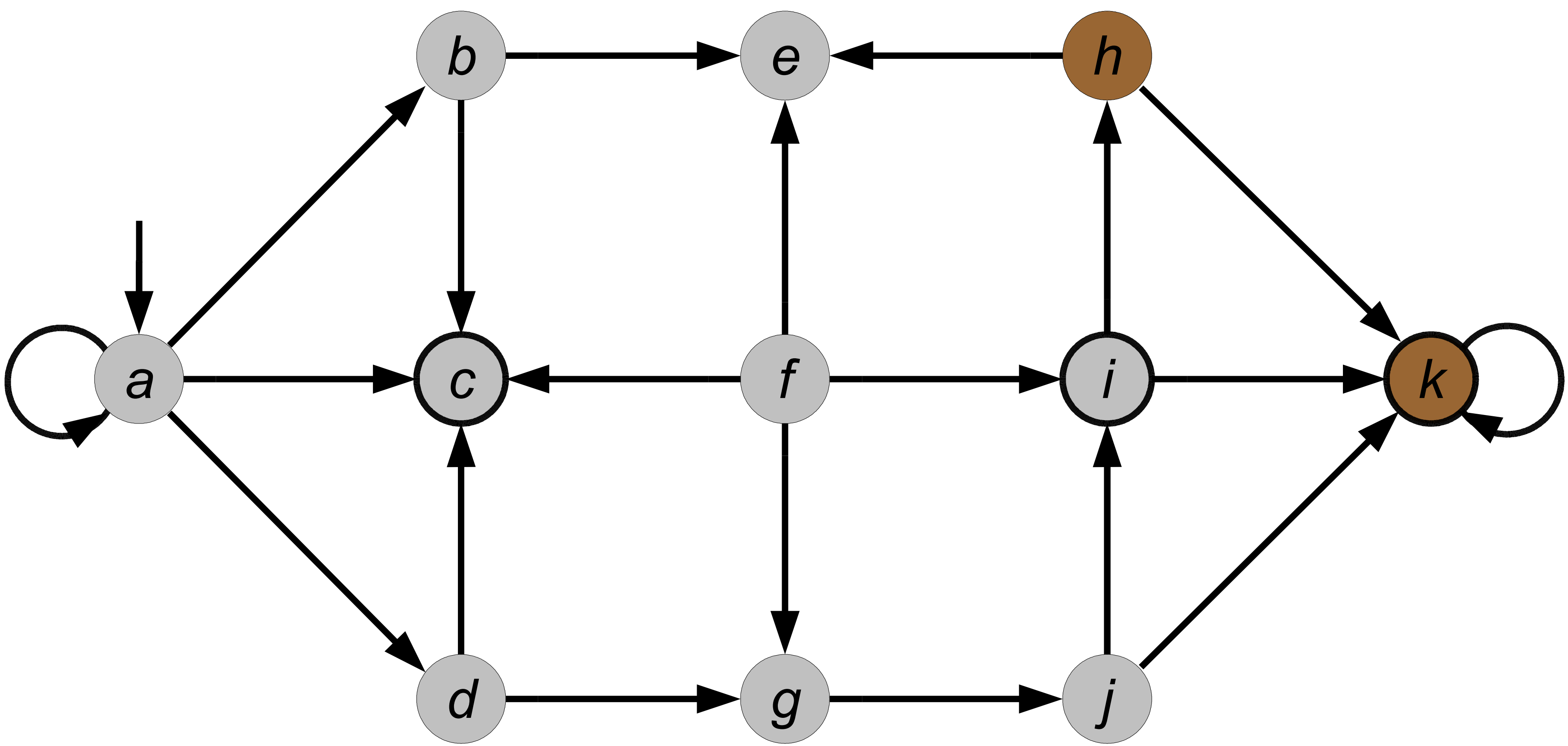
Single Player Game: Buttons and Lights



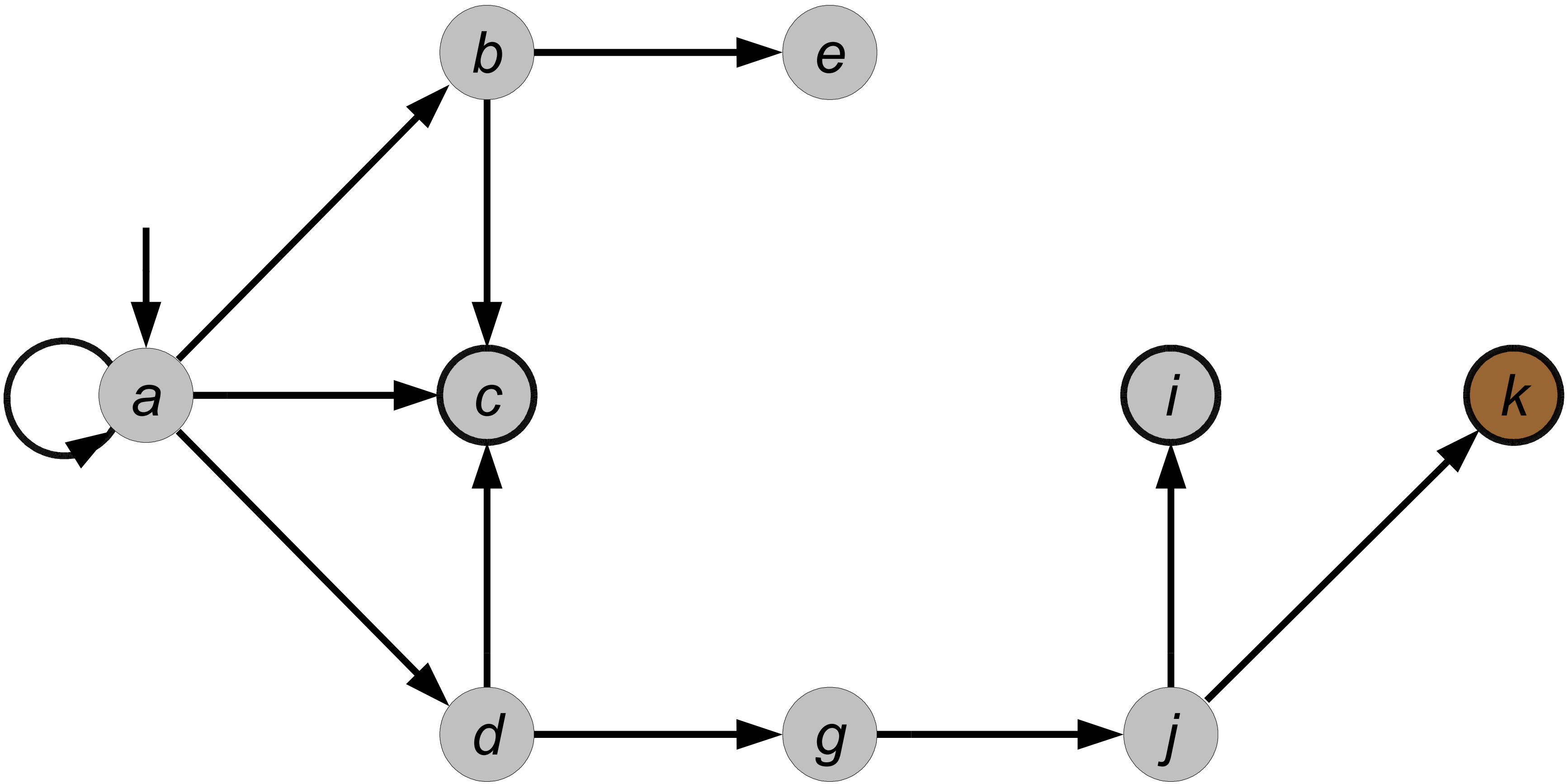
Another Single Player Game: 15-Puzzle



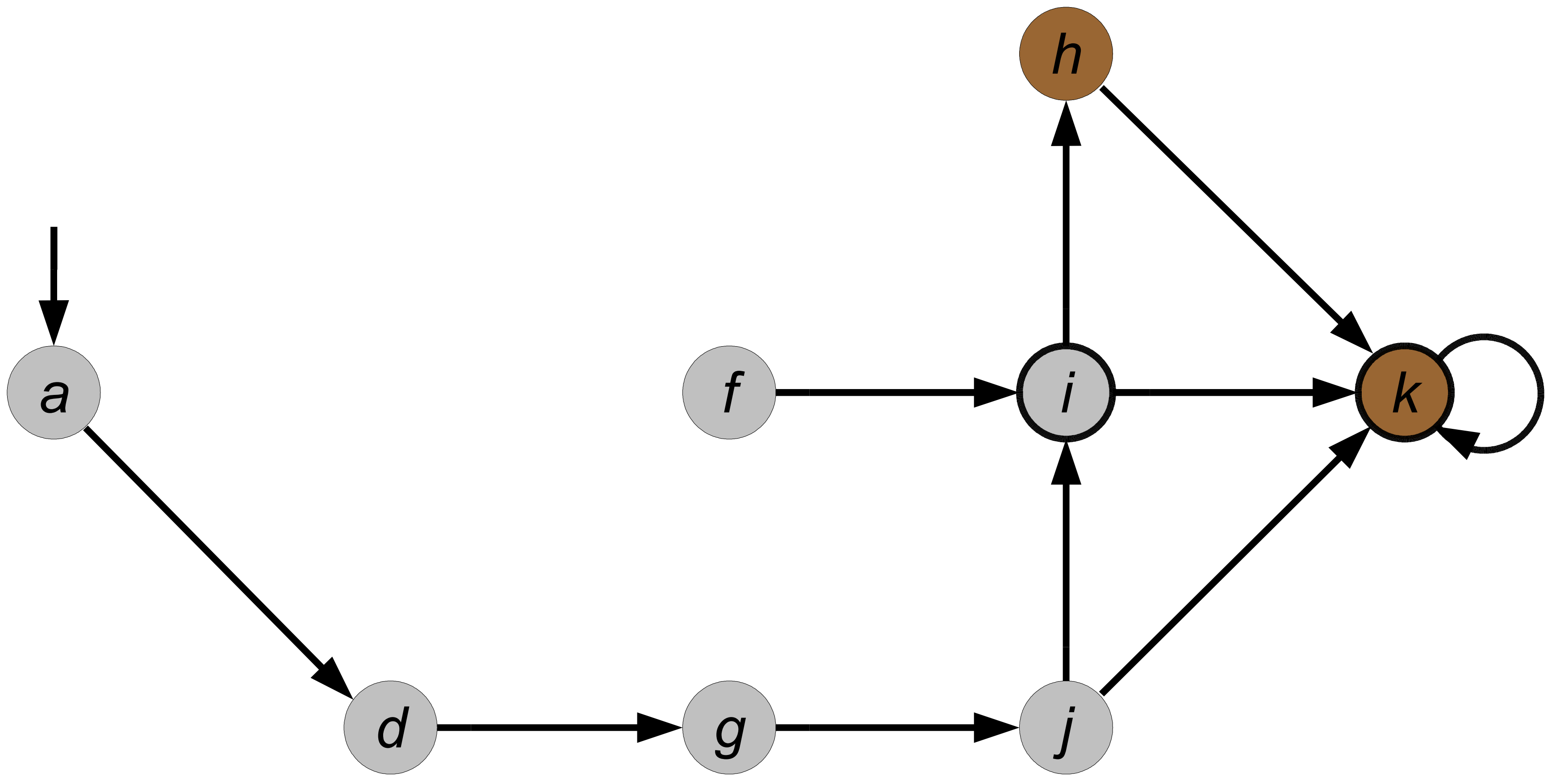
State Machine Model



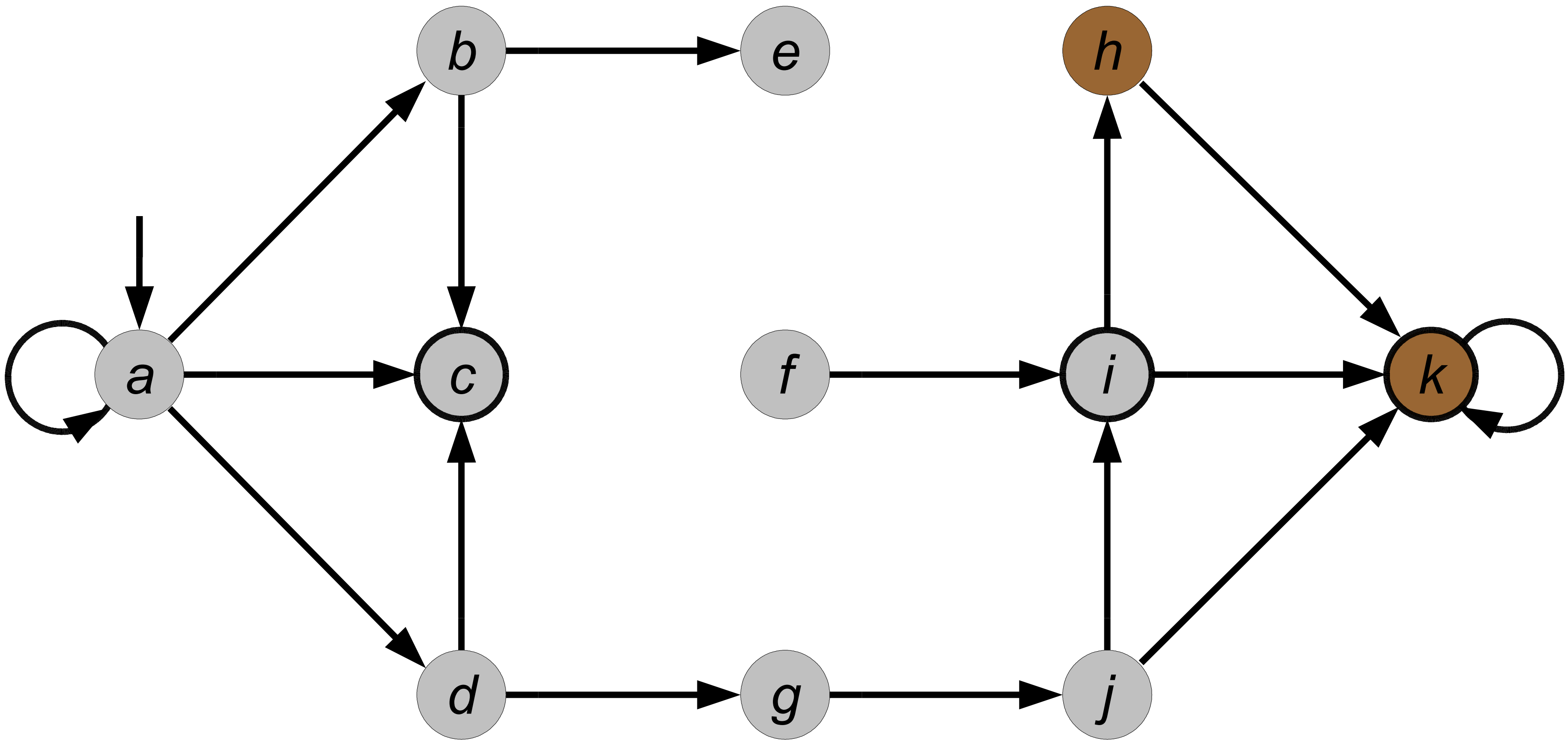
Forward Search



Backward Search

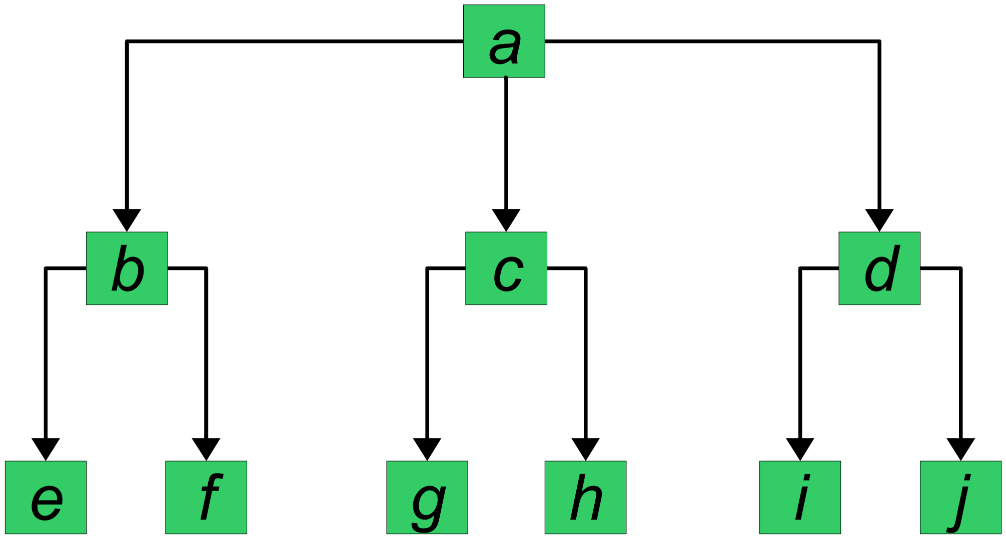


Bidirectional Search



(Note: This is the underlying principle of endgame databases.)

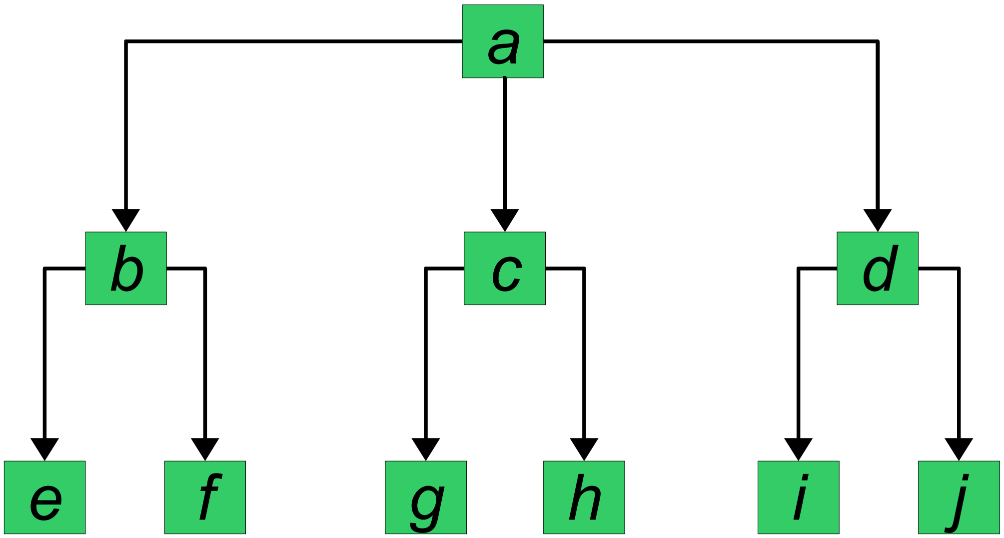
Breadth-First Search



a b c d e f g h i j

Advantage: Finds shortest solution
Disadvantage: Consumes large amount of space

Depth-First Search



a b e f c g h d i j

Advantage: Small intermediate storage
Disadvantage: Susceptible to garden paths
Disadvantage: Susceptible to infinite loops

Time Comparison

Branching factor 2, depth *d*, solution at depth *k*

Time	*Best*	*Worst*
Depth-First	k	$2^d - 2^{d-k}$
Breadth-First	2^{k-1}	$2^k - 1$

Time Comparison

Branching factor *b*, depth *d*, solution at depth *k*

Time	*Best*	*Worst*
Depth-First	k	$\frac{b^d - b^{d-k}}{b-1}$
Breadth-First	$\frac{b^{k-1}-1}{b-1}+1$	$\frac{b^k - 1}{b-1}$

Space Comparison

Worst case for search depth *d* and depth *k*

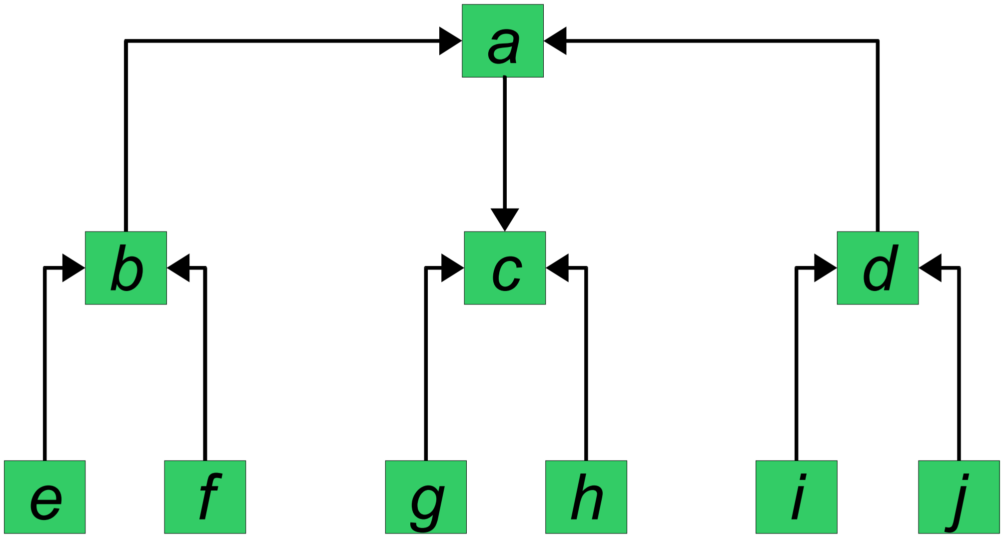
Space	*Binary*	*General*
Depth-First	d	$(b - 1) * (d - 1) + 1$
Breadth-First	2^{k-1}	b^{k-1}

Iterative Deepening

Run depth-limited search repeatedly

- starting with a small initial depth d
- incrementing on each iteration $d := d + 1$
- until success or run out of alternatives

Example



d = 1: *a*
d = 2: *a b c d*
d = 3: *a b e f c g h d i j*

Advantage: Small intermediate storage
Advantage: Finds shortest solution
Advantage: Not susceptible to garden paths
Advantage: Not susceptible to infinite loops

Time Comparison

Worst case for branching factor 2

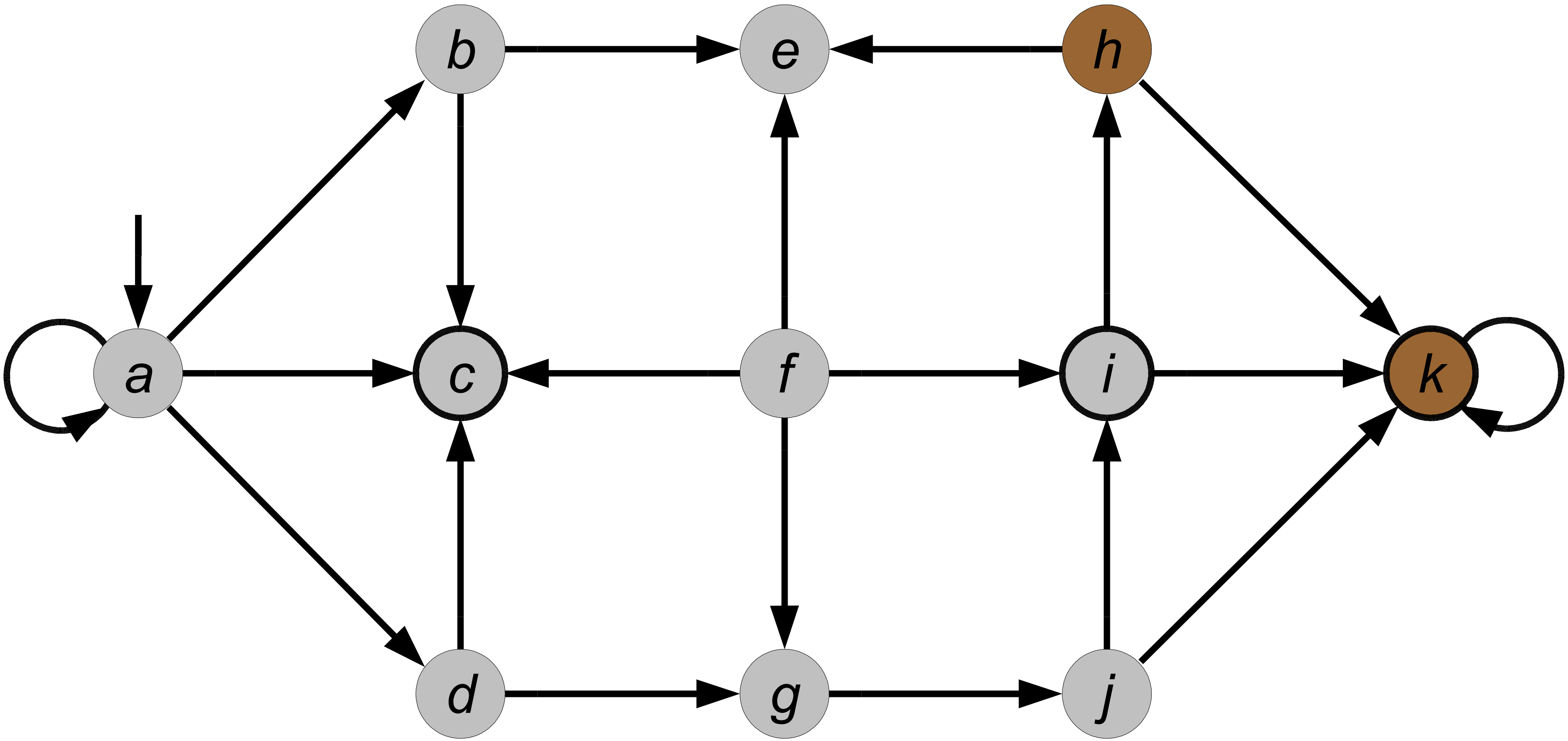
Depth	*Iterative Deepening*	*Depth-First*
1	1	1
2	4	3
3	11	7
4	26	15
5	57	31
n	$2^{n+1} – n – 2$	$2^{n} - 1$

General Results

Theorem: The cost of iterative deepening search is $b/(b\text{-}1)$ times the cost of depth-first search (where b is the branching factor).

Theorem: The space cost of iterative deepening is the same as the space cost for depth-first search.

State Game Graph Model



State-Space Search

A graph can be searched for a path in time linear in the number of nodes.

One small hitch: The graph is implicit in the state description and must be built in advance or incrementally.

Logical Description

```
next(cell(M,N,x)) <=
   does(xplayer,mark(M,N))

next(cell(M,N,o)) <=
   does(oplayer,mark(M,N))

next(cell(M,N,W)) <=
   true(cell(M,N,W)) ∧
   distinct(W,b)

next(cell(M,N,b)) <=
   true(cell(M,N,b)) ∧
   does(P,mark(J,K)) ∧
   (distinct(M,J) ∨ distinct(N,K))
```

A Data Structure for Matches

```
class match (thing)
    {role,          % role of own player
     roles,         % all roles of a game
     theory,        % game description
     startclock,
     playclock,
     hasher,        % hash table
     root,          % root node (initial position)
     fringe         % nodes to be expanded
    }
```

Sample Matches

match 23.

role: **`xplayer`**

roles: **`[xplayer,oplayer]`**

theory: **`[init(cell(1,1,b)),…]`**

startclock: 30

playclock: 30

hasher: *hasharray12*

root: *node1*

fringe: [*node2, node3, node4*]

A Data Structure for Nodes

```
class node (thing)
    {match,
     data,        % current position
     theory,
     parent,      % parent node
     alist,       % list of (action, node) - pairs
     score
    }
```

Sample Node

node2.

match: *match23*

data: [**`true(cell(1,1,x))`**, ...]

theory: [**`init(cell(1,1,b))`**, ...]

parent: *node1*

alist: [(**`mark(1,2)`**,*node21*),(**`mark(1,3)`**,*node22*),...]

score: -1

Basic Subroutines

function *legals (role, node)*

findall(`X`*,* `legal`*(role,*`X`*), node.data ∪ node.theory)*

function *simulate (node,moves)*

findall(`true(P)`*,* `next(P)`*, node.data ∪ moves ∪ node.theory)*

function *terminal (node)*

prove(`terminal`*, node.data ∪ node.theory)*

function *goal (role, node)*

findone(`X`*,* `goal`*(role,*`X`*), node.data ∪ node.theory)*

Node Expansion (Single Player Games)

```
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
```

Time Restriction: Incremental Expansion by Nodes

procedure *incexpand1 (match,count)*
var *node,i*
begin
    for *i* := 1 **until** *i* > *count* **or** *match.fringe* = [ ] **do**
        node := *head(match.fringe);*
        match.fringe := *tail(match.fringe);*
        if *node.score* = -1 **then**
            i := *i* + 1*;*
            match.fringe := *match.fringe* ∪ *expand(node)*
        end-if
    end-for
end

Time Restriction: Incremental Expansion by Time

```
procedure incexpand2 (match,clock)
var node,end
begin
    end := get_universal_time() + clock - 5;
    while get_universal_time() < end and match.fringe ≠ [ ] do
        node := head(match.fringe);
        match.fringe := tail(match.fringe);
        if node.score = -1 then
            match.fringe := match.fringe ∪ expand(node)
        end-if
    end-while
end
```

State Collapse

The game tree for Tic-Tac-Toe has approximately 900,000 nodes. There are approximately 5,000 distinct states. Searching the tree requires 180 times more work than searching the graph.

One small hitch: Recognizing a repeat state takes time that varies with the size of the graph thus far seen. Solution: Hashing

Node Expansion with State Collapse

```
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 := sort(simulate(node,{does(role,a)}));
        if not gethash(data,match.hasher) then
            new := create_node(match,data,node.theory,node,[],-1);
            puthash(data,match.hasher);
            if terminal(new) then new.score := goal(role,new);
            nl := {new} ∪ nl;
            al := {(a,new)} ∪ al
        end-if
    end-for;
    node.alist := al; return nl
end
```

Heuristic Search

These are all techniques for *blind search*. In traditional approaches to game playing, it is common to use *evaluation functions* to assess the quality of non-terminal states.

Example: piece count in chess.

In general game playing, the rules are not known in advance, and an evaluation must be constructed automatically. This is where much of the multifarious intelligence of a general game player lies.

(More on this in Lecture 6.)

Best Move (Single Player Games)

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

Node Evaluation (Single Player Games)

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