Incomplete Search

Game Tree Search
Resource Limitations

Large state spaces
~5000 states in Tic-Tac-Toe
~$10^{30}$ states in Chess

Even larger game trees
~900,000 nodes in Tic-Tac-Toe

Limited Resources
Memory
Time (start clock, move clock)

Programme

Search Direction

Search Strategy

Game Graph Search
Search Direction

Forward Search

Diagram showing the search direction and forward search process.
Backward Search

Bidirectional Search
Search Strategy

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
Breadth First Search

\[ a \quad b \quad c \quad d \quad e \quad f \quad g \quad h \quad i \quad j \]

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

Time Comparison

Branching 2 and depth \(d\) and solution at depth \(k\)

<table>
<thead>
<tr>
<th>Time</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth</td>
<td>(k)</td>
<td>(2^d - 2^{d-k})</td>
</tr>
<tr>
<td>Breadth</td>
<td>(2^{k-1})</td>
<td>(2^k - 1)</td>
</tr>
</tbody>
</table>
**Time Comparison**

Analysis for branching $b$ and depth $d$ and solution at depth $k$.

<table>
<thead>
<tr>
<th>Time</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth</td>
<td>$k$</td>
<td>$\frac{b^d - b^{d-k}}{b-1}$</td>
</tr>
<tr>
<td>Breadth</td>
<td>$\frac{b^{k-1} - 1}{b-1} + 1$</td>
<td>$\frac{b^k - 1}{b-1}$</td>
</tr>
</tbody>
</table>

**Space Comparison**

Total depth $d$ and solution depth $k$.

<table>
<thead>
<tr>
<th>Space</th>
<th>Binary</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth</td>
<td>$d$</td>
<td>$(b-1) \times (d-1) + 1$</td>
</tr>
<tr>
<td>Breadth</td>
<td>$2^{k-1}$</td>
<td>$b^{k-1}$</td>
</tr>
</tbody>
</table>
Iterative Deepening

Run depth-limited search repeatedly,

starting with a small initial depth,

incrementing on each iteration,

until success or run out of alternatives.

Example

```
 a
 a b c d
 a b e f c g h d i j
```

Advantage: Small intermediate storage
Advantage: Finds shortest path
Advantage: Not susceptible to garden paths
Advantage: Not susceptible to infinite loops
Time Comparison

<table>
<thead>
<tr>
<th>Depth</th>
<th>Iterative</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>57</td>
<td>31</td>
</tr>
<tr>
<td>$n$</td>
<td>$2^{n+1} - n - 2$</td>
<td>$2^n - 1$</td>
</tr>
</tbody>
</table>

General Results

Theorem [Korf]: The cost of iterative deepening search is $b/(b-1)$ times the cost of depth-first search (where $b$ is the branching factor).

Theorem: The space cost of iterative deepening is no greater than the space cost for depth-first search.
Incremental Game Tree Search

Game Tree Search

Problem 1: The tree is implicit in the state description and must be built in advance or incrementally.

Problem 2: The clock may be too short to permit complete search.

Solution: Incremental Tree Search
Incremental Game Graph Search

Nodes Versus States

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: The graph is implicit in the state description and must be built in advance or incrementally. Recognizing a repeated state takes time that varies with the size of the graph thus far seen. Solution: Hashing.
Single Player Game Graph

Multiple Player Game Graph
Bipartite Game Graph

Bipartite Association List

Simple Association List

\[ ((a . s2) (b . s3)) \]

Bipartite Association List:

\[ ((a . (((a a) . s2) ((a b) . s1))) ((b a) . s3) ((b b) . s4)))) \]
Incremental Tree Search

Players

class player (thing)
  {var match ← nil;
   var role ← nil;
   var roles ← nil;
   var theory ← nil;
   var startclock ← 0;
   var playclock ← 0
   var hasher ← nil;
   var action ← nil;
   var root ← nil;
   var fringe ← nil}
Tree Expansion

function expands (player,count)
{var node;
 for i ← 1 until i > count or null(player.fringe)
   do {node ← first(player.fringe);
       player.fringe ← rest(player.fringe);
       unless numberp(node.score)
       i ← i + 1;
       player.fringe ← nconc(player.fringe,expand(node))};
 return done}

Nodes

class node (thing)
{var player ← nil;
 var data ← nil;
 var theory ← nil;
 var parent ← nil;
 var alist ← nil;
 var score ← nil}
Single Player Node Expansion

function expand (node)
{var match ← node.match; var role ← match.role;
 var old; var data; var al; var nl;
 for a in legals(role,node)
 {old ← node.data;
  node.data ← cons(does(role, a), old);
  data ← simulate(node);
  node.data ← old;
  new ← makenode(match, data, node.theory, node)
  if termp(new) then new.score ← reward(role, new);
  nl ← cons(new, nl);
  al ← acons(a, new, al)}
 node.alist ← nreverse(al)
 return nreverse(nl)}

Multiple Player Node Expansion

function expand (node)
{var match ← node.match; var role ← match.role;
 var old; var data; var al; var nl;
 for a in legals(role,node)
 {for j in joints(role, a, player.roles, node, nil, nil)
  {old ← node.data;
   node.data ← consactions(match.roles, j, old);
   data ← simulate(node); node.data ← old;
   new ← makenode(match, data, node.theory, node)
   if termp(new) then new.score ← reward(role, new);
   nl ← cons(new, nl);
   bl ← acons(j, new, bl)}
  al ← acons(a, nreverse(bl), al)}
 node.alist ← nreverse(al);
 return nreverse(nl)}
Multiple Player Node Expansion

```plaintext
function expand(node)
{var match ← node.match; var role ← match.role;
  var old; var data; var al; var nl;
  for a in legals(role, node)
    {for j in joints(role, a, player.roles, node, nil, nil)
      {old ← node.data;
        node.data ← consactions(match.roles, j, old);
        data ← simulate(node); node.data ← old;
        new ← makenode(match, data, node.theory, node)
        if termp(new) then new.score ← reward(role, new);
        nl ← cons(new, nl);
        bl ← acons(j, new, bl)
      }
      al ← acons(a, nreverse(bl), al)
    }
  node.alist ← nreverse(al)
  return nreverse(nl)}
```

Subroutines

```plaintext
function consactions(roles, actions, data)
{var nl;
  for role in roles, action in actions
    nl ← cons(does(role, action), nl)
  return nreverse(nl)}

function joints(role, action, roles, node, xl, nl)
{if null(roles) then cons(reverse(xl), nl)
  else if car(roles) = role
    then joints(role, action, cdr(roles), node, cons(action, xl), nl)
  else for x in legals(car(roles), node)
    {nl ← joints(role, action, cdr(roles), node, cons(x, xl), nl)
    return nl}
```
Best Move

function bestmove (node)
{var best;
 var score;
 for pair in node.alist
  do {score ← maxscore(pair.right);
      if score = 100 then return pair.left;
      else if score = 0;
       else if null(best) then best ← pair.left}
 return best or car(node.alist).left}

Node Evaluation

function maxscore (node)
{var score ← nil;
 var max ← 0;
 if node.score then node.score;
 else if null(node.alist) then nil;
 else for pair in node.alist
  do {score ← minscores(pair.right)
      if score = 100 then return 100
      else if not numberp(score) then max ← nil
      else if null(max)
        else if score > max then max ← score}
 return max}
Node Evaluation

function maxscore (node)
{ var score ← nil;
  var max ← 0;
  if node.score then node.score;
  else if null(node.alist) then nil;
  else for pair in node.alist
      do { score ← minscores(pair.right)
        if score = 100 then return 100
        else if not numberp(score) then max ← nil
        else if null(max)
        else if score > max then max ← score }
  return max}

Node Evaluation

function minscores (al)
{ var score ← nil;
  var min ← 100;
  for pair in al
    do { score ← maxscore(pair.right)
      if score = 0 then return 0
      else if not numberp(score) then min ← nil
      else if null(min)
      else if score < min then min ← score }
  return min}
Incremental Graph Search

Node Expansion

```plaintext
function expand (node) {
    var player ← node.player; var old; var data; var al; var nl;
    for a in legals(role, node) {
        for j in joints(player.role, a, player.roles, node, nil, nil) {
            old ← node.data;
            node.data ← consactions(player.roles, j, old);
            data ← sort(simulate(node));
            node.data ← old;
            if new ← gethash(data, player.hasher) then new
            else (new ← makenode(player, data, node.theory, node)
                    gethash(data, player.hasher) ← new;
                    if term(new) then new.score ← reward(player.role, new);
                    nl ← cons(new, nil));
            bl ← acons(j, new, bl)}
        al ← acons(a, nreverse(bl), al)
    node.alist ← nreverse(al)
    return nreverse(nl)}
```
Node Expansion With State Collapse

function expand (node)
{var match ← node.match; var old; var data; var al; var nl;
for a in legals(role,node)
{for j in joints(match.role, a, match.roles, node, nil, nil)
{old ← node.data;
node.data ← consactions(match.roles, j, old);
data ← sort(simulate(node),minlessp);
node.data ← old;
if new ← gethash(data, match.hasher) then new
else (new ← makenode(match, data, node.theory, node)
gethash(data, match.hasher) ← new;
if termp(new) then new.score ← reward(match.role, new);
nil ← cons(new, nil));
bl ←acons(j, new, bl)}
al ←acons(a, nreverse(bl), al)}
node.alist ← nreverse(al)
return nreverse(nl)}

Ordering Code

function minlessp (x,y)
{if numberp(x) then {if numberp(y) then x<y else true}
else if symbolp(x) then {if numberp(y) then nil
else if symbolp(y) then x.name<y.name
else true}
else if numberp(y) then nil
else if symbolp(y) then nil
else for l ← x then cdr(l)
for m ← y then cdr(m)
do {if null(l) then return not null(m)
else if null(m) then return nil
else if minlessp(car(l),car(m)) then return true
else if car(l)=car(m) then nil
else return nil}}}