Search and Games

Adi Botea

ANU Summer Schools in Logic and Learning
February, 2009
Outline

1. Introduction
2. Problem Representation
3. Uninformed Search
4. Informed Search
5. Hierarchical Abstraction
1. Introduction
2. Problem Representation
3. Uninformed Search
4. Informed Search
5. Hierarchical Abstraction
Why Search?

- Lots of real-life applications
  - Navigation, playing games, planning a trip...
- Great tool in many AI areas
  - Planning, scheduling, CSP, SAT, learning...
- One of the hottest research areas in AI
Why Games?

- Fun
- Big industry
- Excellent AI testbed
Our Sample Application

Pathfinding

Find a route on a map, from an initial location to a target location, while avoiding all obstacles.

Pathfinding is useful not only in games but also in robotics, real life (plan a trip)...

In games, search is useful not only in pathfinding, but also in other components. E.g., plan a battle, grab resources, plan non-player character behaviour...
Sample Game Map
Abstract problem into a formal model
- Define state space: states and actions (transitions)
- Define initial state
- Define goal
- Define solutions and their quality measure
Solving a Problem with Search

- Abstract problem into a formal model
  - Define state space: states and actions (transitions)
  - Define initial state
  - Define goal
  - Define solutions and their quality measure
- Use a search algorithm + enhancements

Designing a *good* formal model and a *good* algorithm can be challenging sometimes.
Designing a Formal Model

- Abstract away many details from real life
- Keep relevant details, such that the computed solution is useful
- Ignore irrelevant details, such that the search problem is easier
Traversable Area and Blocked Tiles
State Space
Initial State and Goal (Target)
Assumptions about the Environment

- Static
- Single agent
- Deterministic
- Fully observable
- Finite state space
Our Problem Formulation

States: each location (tile) is a state
Initial state: circle location
Successor function $\text{Succ}(s)$: set of action–state pairs
Given a current state $s$, the successor function tells the resulting state of each action that can be applied in $s$. Each action (up, down, left, right) takes the agent to the corresponding adjacent tile. We consider only straight moves in this lecture.

Goal test: star location
Solution cost: here, number of actions executed
Solution Sequence of actions leading from the initial state to a goal state
Our Problem Formulation

- **States**: each location (tile) is a state
Our Problem Formulation

- **States**: each location (tile) is a state
- **Initial state**: circle location
Our Problem Formulation

- **States**: each location (tile) is a state
- **Initial state**: circle location
- **Successor function** $\text{Succ}(s) = \text{set of action–state pairs}$

Given a current state $s$, the successor function tells the resulting state of each action that can be applied in $s$. Each action (up, down, left, right) takes the agent to the corresponding adjacent tile. We consider only straight moves in this lecture.
Our Problem Formulation

- **States:** each location (tile) is a state
- **Initial state:** circle location
- **Successor function** $\text{Succ}(s) = \text{set of action–state pairs}$

Given a current state $s$, the successor function tells the resulting state of each action that can be applied in $s$. Each action (**up**, **down**, **left**, **right**) takes the agent to the corresponding adjacent tile. We consider only **straight moves** in this lecture.

- **Goal test:** star location
Our Problem Formulation

- States: each location (tile) is a state
- Initial state: circle location
- Successor function \( \text{Succ}(s) = \) set of action–state pairs

Given a current state \( s \), the successor function tells the resulting state of each action that can be applied in \( s \). Each action (up, down, left, right) takes the agent to the corresponding adjacent tile. We consider only straight moves in this lecture.

- Goal test: star location
- **Solution cost**: here, number of actions executed

Solution
Sequence of actions leading from the initial state to a goal state
Search Tree

What Is It?
A space explored by a search algorithm.
Search Tree

What Is It?
A space explored by a search algorithm.

States vs. Tree Nodes
Each node $n$ corresponds to a unique state $s(n)$. The same state can correspond to several nodes.
# Search Tree

## What Is It?
A space explored by a search algorithm.

## States vs. Tree Nodes
Each node $n$ corresponds to a unique state $s(n)$. The same state can correspond to several nodes.

## Generating a Search Tree
The root corresponds to the initial state. Generating the successors (children) of a node $n$:

\[
\forall s' \in \text{Succ}(s(n)), \text{ create a new node } n' \text{ and set } s(n') = s'.
\]
Search Tree

What Is It?
A space explored by a search algorithm.

States vs. Tree Nodes
Each node \( n \) corresponds to a unique state \( s(n) \).
The same state can correspond to several nodes.

Generating a Search Tree
The root corresponds to the initial state.
Generating the successors (children) of a node \( n \):
\[ \forall s' \in \text{Succ}(s(n)), \text{ create a new node } n' \text{ and set } s(n') = s'. \]

Definition
**Expanding** a node means to generate its successors.
Search Tree Example

\[(c, r)\], the current state, contains the column and the row of the unit’s current position.
(c, r), the current state, contains the column and the row of the unit’s current position.
Search Tree Example

\((c, r)\), the current state, contains the column and the row of the unit’s current position.
Search Tree Example

Notice state repetition.
Generic Tree Search Algorithm

Tree-Search
make root node from initial state
initialize Open with root node
loop
  if Open is empty
    return failure
  pop node from Open
  if node contains a goal state
    return corresponding solution
  else
    expand node and add successors to Open
# Search Strategies

## Definition
A strategy is defined by picking the *order of node expansion*.

## Features to Evaluate Strategies
- **Completeness**: does it always find a solution if one exists?
- **Time complexity**: number of nodes generated/expanded
- **Space complexity**: maximum number of nodes in memory
- **Optimality**: does it always find a least-cost solution?

## What Matters for Time and Space Complexity
- $b$, maximum branching factor of the search tree
- $d$, depth of the shallowest solution
- $m$, maximum depth of the state space (may be $\infty$)
Outline

1 Introduction
2 Problem Representation
3 Uninformed Search
4 Informed Search
5 Hierarchical Abstraction
Uninformed Search Strategies

Uninformed strategies use only the information available in the problem definition.

- Breadth-first search
- Depth-first search
- Uniform-cost search
- Depth-limited search
- Iterative deepening search
Breadth-First Search

What: Expand **shallowest** unexpanded node.
How: Keep Open as a **FIFO** queue.
Breadth-First Search

What: Expand **shallowest** unexpanded node.
How: Keep Open as a **FIFO** queue.
Breadth-First Search

What: Expand **shallowest** unexpanded node.
How: Keep Open as a **FIFO** queue.
Breadth-First Search

What: Expand **shallowest** unexpanded node.
How: Keep Open as a **FIFO** queue.
Breadth-First Search

What: Expand **shallowest** unexpanded node.
How: Keep Open as a **FIFO** queue.
Breadth-First Search

What: Expand **shallowest** unexpanded node.
How: Keep Open as a **FIFO** queue.
Properties of Breadth-First Search

Is It Complete?
Yes (if $b$ is finite)

Time $O(b^d + 1)$
Space $O(b^d + 1)$ (keeps every node in memory)

Is It Optimal?
Yes, if cost = 1 per step. Not optimal in general.
### Properties of Breadth-First Search

<table>
<thead>
<tr>
<th>Is It Complete?</th>
<th>Yes (if ( b ) is finite)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>( O(b^{d+1}) )</td>
</tr>
</tbody>
</table>
### Properties of Breadth-First Search

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is It Complete?</td>
<td>Yes (if $b$ is finite)</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^{d+1})$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^{d+1})$ (keeps every node in memory)</td>
</tr>
</tbody>
</table>
## Properties of Breadth-First Search

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is It Complete?</td>
<td>Yes (if ( b ) is finite)</td>
</tr>
<tr>
<td>Time</td>
<td>( O(b^{d+1}) )</td>
</tr>
<tr>
<td>Space</td>
<td>( O(b^{d+1}) ) (keeps every node in memory)</td>
</tr>
<tr>
<td>Is It Optimal?</td>
<td>Yes, if cost = 1 per step. Not optimal in general.</td>
</tr>
</tbody>
</table>
Depth-First Search

What: Expand **deepest** unexpanded node.
How: Open is a **LIFO** queue.
Depth-First Search

What: Expand **deepest** unexpanded node.
How: Open is a **LIFO** queue.
**Depth-First Search**

What: Expand **deepest** unexpanded node.

How: Open is a **LIFO** queue.
Depth-First Search

What: Expand **deepest** unexpanded node.
How: Open is a **LIFO** queue.
Depth-First Search

What: Expand **deepest** unexpanded node.
How: Open is a **LIFO** queue.
Depth-First Search

What: Expand **deepest** unexpanded node.
How: Open is a **LIFO** queue.
Depth-First Search

What: Expand **deepest** unexpanded node.
How: Open is a **LIFO** queue.
Depth-First Search

What: Expand **deepest** unexpanded node.
How: Open is a LIFO queue.
Depth-First Search

What: Expand **deepest** unexpanded node.
How: Open is a **LIFO** queue.
Depth-First Search

What: Expand **deepest** unexpanded node.
How: Open is a **LIFO** queue.
# Properties of Depth-First Search

<table>
<thead>
<tr>
<th>Is It Complete?</th>
</tr>
</thead>
<tbody>
<tr>
<td>No: fails in infinite-depth spaces, such as spaces with loops. If modified to avoid repeated states along path, it is complete.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(b^m)$: terrible if $m$ is much larger than $d$, but if solutions are dense, may be much faster than breadth-first search.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(bm)$, i.e., linear space!</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Is It Optimal?</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
</tr>
</tbody>
</table>

The imagination driving Australia’s ICT future.
Breadth-first versus depth-first search

- Use *breadth-first* search when there exist *short* solutions
- Use *depth-first* search when there exist *many* solutions, or when the depth is bounded and all goal states have maximal depth (e.g., SAT, CSP)
Repeated States

Failure to detect repeated states can turn:

- A linear problem into an exponential one
- A finite state space into an infinite search tree

To handle this, we turn from **tree search** to **graph search**.
Graph-Search

initialize Open with root node (init state)
initialize Closed to empty set
loop
  if Open is empty return failure
  pop node from Open
  if node contains a goal state
    return corresponding solution
  if s(node) not in Closed
    add s(node) to Closed
  expand node and add successors to Open
Outline

1. Introduction
2. Problem Representation
3. Uninformed Search
4. Informed Search
5. Hierarchical Abstraction
Why Should We Use Informed Search?

Often, they find a solution much more quickly than blind (uninformed) search.
Informed Search

Why Should We Use Informed Search?

Often, they find a solution much more quickly than blind (uninformed) search.

- Expand with priority nodes that look more promising.
- Use a *heuristic evaluation function* to guide the search.
Evaluation Function

- $g(n)$: exact cost of reaching $n$ from the root node.
- $h(n)$: estimated cost of reaching a goal node from $n$. This is called a heuristic function.
- $f(n) = g(n) + h(n)$: estimated cost of a solution (path from root to goal) that contains $n$.

Typical evaluation functions use $g$ or $h$ or $f$. 

Path from root to $n$. $g(n)$ is the path cost

$h(n)$ is the estimated cost of a path from $n$ to a goal node
Best-First Search

What Is It

Generic search algorithm that expands the most promising node first.

How It Does It

Keep the Open list sorted according to an evaluation function. Select for expansion a node with the best (smallest) evaluation.

Depending on how the evaluation function is defined, several variations of best-first search can be designed. (Examples will come next).
Variations of Best-First Search

- A* [6]: Uses \( f(n) \) as an evaluation function

A* – de facto standard in pathfinding search [1]
Variations of Best-First Search

- A* [6]: Uses $f(n)$ as an evaluation function

A* – de facto standard in pathfinding search [1]

- Weighted A*: The evaluation function is $g(n) + W \times h(n)$, $W > 1$
Variations of Best-First Search

- A* [6]: Uses $f(n)$ as an evaluation function

  A* – de facto standard in pathfinding search [1]

- Weighted A*: The evaluation function is $g(n) + W \times h(n)$, $W > 1$

- **Uniform-cost search (uninformed)**: The evaluation function is $g(n)$
Variations of Best-First Search

- **A* [6]:** Uses $f(n)$ as an evaluation function

A* – de facto standard in pathfinding search [1]

- Weighted A*: The evaluation function is $g(n) + W \times h(n), W > 1$
- Uniform-cost search (uninformed): The evaluation function is $g(n)$
- **Greedy search** (aka pure heuristic search): The evaluation function is $h(n)$
Where Does $h$ Come From?

A Very General Strategy To Compute $h(n)$

- Relax the original problem.
- Solve the relaxed problem. I.e., find a path from $n$ to a goal.
- Use the relaxed solution as an estimate for the real solution. I.e., define $h(n)$ as the length of a relaxed path from $n$ to a goal.

There can be a trade-off between the speed of computing $h$ and its accuracy.

General Assumption
For any goal state $G$, $h(G) = 0$. 
A simple, effective and fast to compute heuristic
Relaxation: Assume that there are no obstacles between a current node \( n \) and the target (goal).

\[ h(n) = |x_n - x_T| + |y_n - y_T| \]
Admissible Heuristics

Definition
A heuristic $h$ is admissible if $h(n) \leq h^*(n), \forall n$, where $h^*(n)$ is the exact shortest distance from $n$ to a goal. $h^*$ is called the perfect heuristic.

Why Are They Useful?
Search algorithms such as A* are optimal when $h$ is admissible.
Consistent Heuristics

Definition
A heuristic $h$ is consistent if $h(n) \leq c(n, n') + h(n')$, where: $n'$ is a successor of $n$, and $c(n, n')$ is the cost of a transition from $n$ to $n'$. Also, recall that $h(G) = 0$ for every goal node $G$.

Property
If $h$ is consistent, then $f(n)$ cannot decrease along a path in the search graph.

Property
A consistent heuristic is admissible.
A* Example

Dark blue: obstacle. Number: \( f(n) = g(n) + h(n) \).
Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: $f(n) = g(n) + h(n)$. Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: \( f(n) = g(n) + h(n) \).
Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: $f(n) = g(n) + h(n)$.
Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: $f(n) = g(n) + h(n)$. Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: $f(n) = g(n) + h(n)$. Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: \( f(n) = g(n) + h(n) \).
Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: \( f(n) = g(n) + h(n) \).
Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: $f(n) = g(n) + h(n)$. Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: \( f(n) = g(n) + h(n) \).
Green: open list. Yellow: closed list.
Dark blue: obstacle. Number: \( f(n) = g(n) + h(n) \). Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: \( f(n) = g(n) + h(n) \).
Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: $f(n) = g(n) + h(n)$. Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: \( f(n) = g(n) + h(n) \).
Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: \( f(n) = g(n) + h(n) \).
Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: \( f(n) = g(n) + h(n) \).
Green: open list. Yellow: closed list.
A* Example

Dark blue: obstacle. Number: $f(n) = g(n) + h(n)$.
Green: open list. Yellow: closed list.
A* Example

Is It Complete?
Yes, unless there are infinitely many nodes \( n \) with 
\[ f(n) < f(\text{goal}). \]
# Properties of A*

<table>
<thead>
<tr>
<th>Is It Complete?</th>
<th>Yes, unless there are infinitely many nodes $n$ with $f(n) &lt; f(\text{goal})$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>$O(b^{d+1})$. In practice, often better than uninformed search.</td>
</tr>
</tbody>
</table>
Properties of A*

Is It Complete?
Yes, unless there are infinitely many nodes $n$ with $f(n) < f(\text{goal})$.

Time
$O(b^{d+1})$. In practice, often better than uninformed search.

Space
$O(b^{d+1})$ (keeps every node in memory).
Properties of A*

Is It Complete?
Yes, unless there are infinitely many nodes \( n \) with \( f(n) < f(\text{goal}) \).

Time
\( O(b^{d+1}) \). In practice, often better than uninformed search.

Space
\( O(b^{d+1}) \) (keeps every node in memory).

Is It Optimal?
Yes, if \( h \) is admissible.
Is A Good Heuristic Enough?

Questions
1) How close are $h$ and $h^*$?
2) How many locations does A* expand?
Q1: How close are $h$ and $h^*$?

$|h^*(n) - h(n)| \leq 2, \forall n$. $h$ is almost perfect here.
Q2: How many locations does A* expand?

**Almost all**: all yellow + part of the white ones.
Not necessarily, as shown earlier
More or less similar behaviour has been shown for AI planning domains by Helmert and Röger [7]
In pathfinding, we deal with this by abstracting the state space into a smaller representation [3, 5, 10]
Outline

1. Introduction
2. Problem Representation
3. Uninformed Search
4. Informed Search
5. Hierarchical Abstraction
HPA*: A Hierarchical Pathfinding Algorithm

- Much faster path computation than low-level A*
- Better *first-move lag* than low-level A*
- Optimality is lost, but solutions are nearly optimal in practice
- Abstraction requires no human input
- Works for many topologies (e.g., building interiors, forests, landscapes...)
How Does HPA* Work?

- **Preprocessing (once per map)**
  - Build an abstract search graph, much smaller than the initial one.

- **Runtime processing**
  - **Start and target integration**: Insert start and target into the abstract graph.
  - **Abstract search**: Run A* on the abstract graph to find an abstract solution quickly.
  - **Path refinement**: Refine portions of an abstract solution as needed. This may require additional small A* searches or may use cached information. Notice the memory vs. time trade-off.
Split map into clusters (e.g., 10x10 tiles per cluster).
Identify entrances between adjacent clusters.
Preprocessing

Define one transition (two abstract nodes + one abstract inter-cluster edge) per entrance.
Define intra-cluster edges to connect abstract nodes inside the same cluster.
Only 44 abstract nodes. The low-level graph has almost 1,200 nodes.
Add start and target as abstract nodes.
Connect start and target to the rest of the abstract graph.
Run A* on abstract graph. Find an abstract path such as the pictured one.
Map abstract edges back into detailed path segments. Do it only when/if needed.
Map abstract edges back into detailed path segments. Do it only when/if needed.
Path Refinement

Map abstract edges back into detailed path segments. Do it only when/if needed.
Tested on maps from Baldur’s Gate (Bioware)
Maps have up to 320x320 tiles
HPA* is up to 10x faster than A*. The speed-up is expected to increase on larger maps.
Solution quality is about 1% off optimal after a simple post-processing step (not discussed in this lecture).
Conclusion

- Search: powerful tool with lots of applications
- Hot research area
- Lots of fun! Interested?
Part of this material follows the structure of the slides available at http://aima.cs.berkeley.edu/, that come with the textbook “Artificial Intelligence – A Modern Approach” by Russell and Norvig.
Yngvi Björnsson, Markus Enzenberger, Robert Holte, and Jonathan Schaeffer.

Yngvi Björnsson and Kári Halldórsson.
Improved Heuristics for Optimal Path-finding on Game Maps.

Adi Botea, Martin Müller, and Jonathan Schaeffer.
Near Optimal Hierarchical Path-Finding.


Peter Hart, Nils Nilsson, and Bertram Raphael.
A Formal Basis for the Heuristic Determination of Minimum Cost Paths.


David Silver. Cooperative Pathfinding.

