I2AI: Lecture 03

Solving problems by search + games and adversarial search

Lubica Benuskova

Reading: AIMA 3rd ed. chap. 3.1 – 3.5.2, chap. 5.1 – 5.4, chap. 17.5.1 – 17.5.2

Goal-based agents

Goals

- Intelligent agents are supposed to maximise their performance measure. Learning helps – but is that all to it?
- Intelligent agents should be able to adopt a goal. How can it help?
- Courses of actions that do not reach the goal can be rejected without further consideration.
- Goal formulation based on the current situation and the agent's performance measure, is the 1st step in problem solving.

Problem

- Obviously, the goal is related to the problem we want to solve.
- A problem can be divided into these parts:
  - The initial state of the agent and the environment;
  - A set of actions agent can perform;
  - A transition model describing the results of those actions;
  - A state space: the set of all possible states of the agent and environment;
  - A goal test function (tells us whether the goal has been achieved);
  - Path cost function that tells us the cost of the solution;
  - Solution, which is a concrete path through the state space from the initial state to a goal state.

Real-world problems

- Consider the airline travel problem for the travel-planning Web site:
  - Initial state: this is specified by the user (Vienna, 1/5/16 @10am);
  - Actions: Take any flight from the current location, in economy class, leaving at or after current time, leaving enough time for within-airport transfers, customs, check-ins, etc;
  - Transition model: the state resulting from taking a flight will have the flight's destination and arrival time as the current location and current time, respectively;
  - State space: includes airports and times of all flights of all airlines;
  - Goal test: am I in Sydney before my friend's birthday party?
  - Path cost: this depends on monetary cost of each flight, waiting time, flight time, customs and immigration procedures, etc.
  - Solution: a path that gets me to Sydney at minimum cost on time.

Route-finding problems

- Route-finding problem is defined in terms of specified locations and transitions along links between them.
- Route-finding algorithms are used in variety of applications, e.g.
  - Web sites for planning (travel, study, building a house, etc.);
  - In-car systems that provide driving directions;
  - Routing video (and other) streams in computer networks;
  - Robot navigation;
  - Automatic assembly sequencing of intricate machines;
  - VLSI layout, positioning millions of components on a chip to minimise the area, circuit delays, manufacturing cost, etc.
Searching for solutions

- The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the **branches** (treny) are actions and **nodes** (uzly) correspond to states in the state space of a problem.

- The root of the tree is the initial state. The 1st step is to test whether this is the goal state.

- Then we need to consider various actions. We do this by **expanding** the current state by applying each legal action to the current state, thereby **generating** a new set of states.

---

The essence of search

- The essence of the search along the tree of possible branches is to **follow one option now** and put the others aside for later in case the first path does not lead to a solution.

- In every case we add branches from the parent node leading to one or more child nodes.

- Each of these child nodes available for further expansion is called a **frontier** (treny) (Some call it the open list.)

- In every case we add branches from the parent node leading to one or more **child nodes** (dzěnske uzly) or **successors**.

---

Illustration of node expansion

- First, only root (black circle) has been expanded into four child nodes (white circles), which are called a **frontier** (treny). The frontier always separates the explored region from the unexplored region (grey nodes).

- Then, only one leaf (the right one) has been expanded into its three child nodes.

- The remaining successors of the root have been expanded in a clockwise order.

---

Order of expanding the nodes

- The process of expanding the nodes on the frontier continues until either a solution is found or there are no more states to expand (or we run out of time...).

- All (tree) search algorithms share this basic strategy; they vary primarily according to how they choose **which node to expand next** – the so-called **search strategy**.

---

Book-keeping structure for a node

- Search algorithms need a structure to keep track of the search tree that is being constructed. For each node \( n \) of the search tree we have a data structure that contains four main parts:

  - **n.state**: the state in the state space which the node represents;
  - **n.parent**: the node that generated this node;
  - **n.action**: the action that was applied to the parent to generate this node;
  - **n.path-cost**: the cost of the path from the initial state to the node, as indicated by the parent pointers.

---

Book-keeping structure for the frontier

- We need to store the frontier in such a way that the search algorithm can easily choose the next node to expand according to its strategy.

  - The appropriate data structure for this is a **queue** (treny, Horton, rad).
  - The operations on the queue are these:
    - **EMPTY** (queue) returns true only if there are no more items in it;
    - **POP** (queue) removes the first item of the queue and returns it;
    - **INSERT** (item, queue) inserts an item and returns the resulting queue.

  - Queues are characterised by the order in which they store the inserted items: **FIFO** (pops the oldest item), **LIFO** or **stack** (pops the newest one) and **priority queue** (pops the highest priority item).
Measuring performance

- We can evaluate the search algorithm performance in four ways:
  - **Completeness**: is the algorithm guaranteed to find a solution when there is one?
  - **Optimality**: does the strategy find the optimal solution in terms of pre-defined criterion (e.g., cost of the air ticket)?
  - **Time complexity**: how long does it take to find a solution?
  - **Space complexity**: how much memory is needed to perform the search?

- The way to avoid exploring redundant paths is to remember where one has been. So, we always have a data structure called the **explored set** (or closed list) of nodes.

Classification of search strategies

- **The uninformed (or blind) search**: strategies that have no additional information about states beyond that given at the start. All they can do is to generate child nodes and distinguish a goal from a no-goal.
  - Breadth-first search (prehľadávanie do šírky)
  - Uniform-cost search (prehľadávanie s jednotnou cenou)
  - Depth-first search (prehľadávanie do hloubky)

- **Informed (or heuristic) search**: strategies that “know” whether one non-goal state is “more promising” than another.
  - Best-first search (prehľadávanie prvého najlepšieho)
  - Greedy best-first search (lačné prehľadávanie prvého najlepšieho)
  - A* search (algorithmus A*)

Breath-first search

- The root node is expanded first. Then all the successors of the root node are expanded next, one by one, then their successors, etc.
- In general, all the nodes at a given depth in the search tree are expanded before the next level are expanded.

- FIFO queue is used: new nodes go back of the queue, and old nodes get expanded first. The goal test is applied to each node when it is generated rather when it is selected for expansion.

Uniform-cost search

- The root node is expanded first. Then all the successors of the root node are generated — but here comes the difference:

- Modification of BFS: instead of expanding all the shallowest nodes, UCS expands the node n with the lowest past cost $g(n)$ so far.

- This is done by ordering the queue of nodes according to path cost $g(n)$.

- Another modification is that the goal test is applied when the node is selected for expansion, rather when it is generated (saves time).

- Uniform-cost search does not care about the number of steps a path has, but only about its total cost.

Depth-first search

- DFS always expands the deepest node in the current frontier of the search tree.

- LIFO queue is used: the most recently generated node is chosen for expansion.

Modifications of DFS

- **Backtracking**: uses less memory, because only one successor is generated at a time rather than all successors — each partially expanded node remembers which successor to generate next.

- **Depth-limited search**: deals with the problem of infinite (or very large) search space. We simply limit the depth of the search by imposing the maximal depth for search (which is informed by the knowledge of the problem).

- **Iterative-deepening DFS**: it is a combination of DFS and BFS that finds iteratively the best depth limit. It does so by gradually increasing the limit — first 0, then 1, then 2, etc. — until a goal is found.
Evaluation of uninformed searches

- Symbols: $b =$ branching factor; $d =$ depth of shallowest solution; $m =$ maximum depth of the search tree; $l =$ depth limit. Superscripts notes:
  1 complete if $b$ is finite; 2 complete if step cost $C > e > 0$; 3 optimal if step costs are all equal; $C^*$ optimal path cost.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>BFS</th>
<th>UniformDFS</th>
<th>DFS</th>
<th>Limit-depthDFS</th>
<th>IterativeDFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^d)$</td>
<td>$O(b^{d-C^*/e})$</td>
<td>$O(b^l)$</td>
<td>$O(b^l)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^d)$</td>
<td>$O(b^{d-C^*/e})$</td>
<td>$O(bm)$</td>
<td>$O(bd)$</td>
<td>$O(bd)$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Uninformed search – summary

- **Breadth-first search** expands the shallowest nodes first; it is complete, optimal, but has exponential space complexity.

- **Uniform-cost search** expands the node with lowest path cost, $g(n)$, and is optimal for general step costs.

- **Depth-first search** expands the deepest unexpanded node first. It is neither complete nor optimal, but has linear space complexity.
  - **Depth-limited search** adds a depth bound.

- **Iterative deepening search** calls depth-first search with increasing depth limits until a goal is found. It is complete, optimal, has time complexity comparable to breadth-first search, and has linear space complexity.

Informed search strategies

- These strategies use a problem-specific knowledge to guide the search in addition to uninformed search algorithms.

  - Thus, the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost to the node $n$ + some problem-specific heuristic evaluation function $h(n)$.
  - The general approach is called the **best-first search**, in which a node is selected for expansion based on $h(n)$:
    - **Definition**: $h(n) =$ estimated cost of the cheapest path from the state at node $n$ to a goal state.
  - The node, for which the value of $h(n)$ is minimal is expanded first. If $n$ is the goal node then $h(n) = 0$.

Greedy best-first search

- This strategy tries to expand the node that is closest to the goal. Thus, it evaluates nodes by using only the heuristic function, i.e. $f(n) = h(n)$.

- Let us illustrate the greedy BFS on the example of flying from Vienna to Sydney (we need to know the distances between all the airports on all the possible routes):
  - Generate the child nodes of the Vienna node;
  - then choose for expansion the next international airport (child node), that is closest to Sydney, let say Dubai;
  - Then generate all the successors of Dubai en route to Sydney;
  - then expand the closest one to Sydney, let’s say Bangkok, etc.

- The algorithm is greedy $b/c$ at each step it tries to get as close as possible to the goal as it can. The worst-case complexity is $O(b^n)$.

$A^*$ search

- The value of evaluation function $f(n)$ at the node $n$ is the sum of the cost to reach the node $g(n)$ and the cost from the node to the goal $h(n)$:
  - $f(n)$ is the estimated cost of the cheapest solution through $n$.

  - Thus, the node that is going to be expanded at each level is the node with lowest value of $f(n)$.

  - It is basically the same as the uniform-cost search, except that $A^*$ uses $f(n) = g(n) + h(n)$, and not just the cost $g(n)$.

  - The algorithm is both complete and optimal.

  - The complexity is $O(b^{d-e})$, where $d$ is the solution depth and $e \equiv (h^*-b)/h^*$, where $h^*$ is the actual cost of the solution.
Informed search – summary

- **Best-first search** algorithm selects a node for expansion according to the value of evaluation function \( f(n) = g(n) + h(n) \).

- **Greedy best-first search** expands nodes with minimal \( h(n) \), while ignoring \( g(n) \). It is not optimal but is often efficient.

- **A* search** expands nodes with minimal \( f(n) = g(n) + h(n) \). A* is complete and optimal. The space complexity of A* is still prohibitive.

- **IDA* algorithm**: The simplest way to reduce memory requirements of the A* search is to adapt the idea of iterative deepening, resulting in the iterative-deepening A* (IDA*) algorithm.

Adversarial search (prehľadávanie s protivníkom)

- Next, we will look at multi-agent situations, in which each agent needs to consider the actions of other agents and how they affect its own welfare.

- We cover competitive environments, in which the agents’ goals are in conflict, giving rise to adversarial search problems—often known as games.

- In AI, the most common games are deterministic, turn-taking, 2-player, zero-sum or perfect-information games (such as chess).

- We assume having a fully observable, deterministic environment, in which agents take turns to act, and the utility values at the end are always equal and opposite (hence zero sum).

Chess

- Let’s consider chess: it has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has \( 35^{100} \approx 10^{154} \) nodes. (Imagine a tree search algorithm for this!

- Games in the real-world therefore require the ability to make some decision when calculating the optimal decision is not feasible.

- We begin with a definition of an optimal move and an algorithm for finding it even when time is limited.

Game as a search problem

- The initial state \( S_0 \), which specifies how the game is set at the start.

- **PLAYER(S)**: defines which player has to move in a state \( s \).

- **ACTIONS(s)**: defines the set of legal moves in a state \( s \).

- **RESULT(s, a)**: the transition model which defines the results of a move.

- **TERMINAL-TEST(s)**: returns true if the game is over else returns false.

- **UTILITY(s, p)**: a utility function (also called an objective function or payoff function) defines a numerical value for a game that ends in terminal state \( s \) for a player \( p \).

  - In chess the outcome is win, loss, or draw, with values 1, 0, or \( \frac{1}{2} \).

- The initial state, ACTIONS function and RESULT function define the game tree for the game—a tree where the nodes are the game states and edges are moves. Let us know consider an example of tic-tac-toe.

MIN, MAX and tic-tac-toe

- Let’s have two players, called **MAX** and **MIN**. MAX starts the game and has nine possible moves from the initial state. Play alternates between MAX placing an X and MIN placing an O, until we reach a terminal state.

- Terminal state: either one of the players has three in a row or all the squares are filled. The final value 1 is good for MAX and bad for MIN.

Optimal decision in games

- In a normal search problem, the optimal solution would be a sequence of actions leading to a goal with the maximal benefit (minimal cost).

- In adversarial search, MAX must find a strategy that specifies MAX’s move from the initial state and then MAX’s moves in the states resulting from every possible response by MIN.

  - Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

  - Even the whole tree for tic-tac-toe is too big (9! = 362,880 possible terminal nodes) so we illustrate the idea on a simpler game.
A two-ply game tree

• Instead, we will use this trivial game: its tree is only one move deep, consisting of two half-moves, each of which we call a ply.
• The possible moves for MAX at the root node are labeled $a_1$, $a_2$, and $a_3$. The possible replies to $a_i$ by MIN are labeled $b_1$, $b_2$, $b_3$, and so on.
• The game ends after one move by each MAX and MIN.

Max prefers max values, min prefers min values

• The utilities of the terminal states range from 2 to 14.
• Given a choice, MAX prefers to move to a state of maximum value and MIN prefers a state of minimum value.
• We need some quantification of the optimal strategy. This is provided by the so-called minimax function $\text{MINIMAX}(n)$.

Minimax value

• Definition: the minimax value of the node $n$ is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from that node to the end of game. The minimax value of the terminal state is just its utility.

\[
\text{MINIMAX}(n) = \begin{cases} 
\text{UTILITY}(s) & \text{if TERMINAL}(s) \\
\max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MAX} \\
\min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MIN}
\end{cases}
\]

• Actions($s$) is the set of all actions in state $s$; and $a$ is a concrete action.

• Let’s apply these definitions to the two-ply game.

The MINIMAX algorithm

• The MINIMAX algorithm performs a complete depth-first search in the search tree.

• The algorithm computes the minimax decision from the current state always maximising the utility for MAX and minimising it for MIN.

Alpha-beta pruning

• The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.

• Improvement: a node for expansion is chosen based on the actual MIN’s move, thus we do not have to look at every node in the tree.

– Alpha = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
– Beta = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

• Alpha-beta search updates the values of alpha and beta as it goes along and prunes the remaining branches at a node.
State-of-the-art chess programs

- IBM's DEEP BLUE chess program, defeated world champion Garry Kasparov in 1997. For the first time in history AI beat a human in chess.
- Deep Blue ran on a parallel computer with 30 IBM RS/6000 processors doing alpha-beta search + 480 custom VLSI chess processors that performed move generation and move ordering.
- Deep Blue searched up to 30 billion positions per move, reaching depth 14 routinely. In some cases the search reached a depth of 40 plies.
- The evaluation function had over 8000 features, many of them describing highly specific patterns of pieces.
- An “opening book” of about 4000 positions was used, as well as a database of 700,000 grandmaster games and a database of endgames.
- HYDRA, the successor to DEEP BLUE. HYDRA runs on a 64-processor cluster with 1 gigabyte per processor, hundreds of custom FPGA chips, reaches 200 million evaluations /s, and 18 plies deep.

An interesting twist

- Let us assume that all players take actions simultaneously, i.e. at each round, no player has knowledge of other players' choices.
- The game theory of these types of “games” has serious implications for decision-making situations including: auctioning the oil drilling rights, wireless frequency spectrum rights, product development and pricing decisions, and all kinds of economic and military decisions.
- First, let us consider single-move game defined by these components:
  - Players or agents that will be making decisions.
  - Actions that the players can choose from.
  - A payoff function that gives the utility to each player for each combination of actions by all the players.

Prisoner’s dilemma (vážňova dilema)

- Two alleged burglars, Alice and Bob, are caught red-handed near the scene of burglary and are interrogated separately. A prosecutor offers each a deal if you testify against your mate, you’ll go free for being a cooperative one, while your mate will serve 10 years in prison. However, if you both testify against each other, you both get 5 years. Alice and Bob also know that if both refuse to testify, they will get only 1 year each for possessing a stolen property (a lesser offence).
- To help each reach a rational decision they both construct this payoff matrix:

<table>
<thead>
<tr>
<th>Alice: testify</th>
<th>Alice: refuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob: testify</td>
<td>A=−5, B=−5</td>
</tr>
<tr>
<td>Bob: refuse</td>
<td>A=−10, B=0</td>
</tr>
</tbody>
</table>

Dominant strategy

- Each of them analyses the payoff matrix as follows: “Suppose my mate testifies. Then I get 5 years if I testify too and 10 years if I don’t, so in that case testifying is better. On the other hand, if my mate refuses to testify, then I get 0 years if I testify and 1 year if I refuse, thus in that case testifying is again better. In either case, it’s better for me to testify.”
- Each player has discovered that testify is the so-called dominant strategy for this game. Being rational ≡ to play the dominant strategy.
- We say that a strategy s dominates other strategy s’ if the outcome of s is better for a given player than the outcome of s’ for every choice of strategies played by other players.

Pareto optimality and Nash equilibrium

- We say that an outcome is Pareto optimal if there is no other outcome that all players would prefer. (Named after Wilfredo Pareto)
- An outcome is Pareto dominated by another outcome if all players would prefer the other outcome.
- When each player has a dominant strategy, the combination of these strategies is called a dominant strategy equilibrium.
  - In general, a strategy profile forms an equilibrium if no player can benefit by switching strategies, given each player sticks with the same strategy.
- The mathematician John Nash proved that every game has at least one equilibrium, hence it is called Nash equilibrium.

The dilemma

- The dilemma in the prisoner’s dilemma is that the equilibrium outcome is worse for both players that the outcome they would get if they both refused to testify.
- In other words, (testify, testify) is Pareto dominated by the outcome of (refuse, refuse) strategy.
- Strategy (refuse, refuse) is allowed, but it is hard to see how the rational agents would get there, given our definition of the game.
- Each player contemplating playing refuse will reckon that he/she would be on a safe side by playing testify – that is an attractive power of an equilibrium point.
Tit-for-tat (oko za oko) in a repeated game

- Let us suppose that the players face the same choice repeatedly many times, but each with the knowledge of the history of all players’ previous choices. Another rule is that payoffs add over time.

- So, will Alice and Bob refuse to testify knowing they will meet again?

- The most famous strategy in this situation is called tit-for-tat. It starts with (refuse, refuse) and then imitating the other player’s previous move on all subsequent moves.

- So, Alice would refuse as long as Bob refuses and would testify the move after Bob testified, but would go back to refusing if Bob did.

Adversarial search – summary

- In two-player zero-sum games with perfect information, when players take alternative turns, the MINIMAX algorithm can select optimal moves by a depth-first enumeration of the game tree.

- The alpha–beta search algorithm computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.

- The most famous of games when both players play simultaneously is Prisoner’s dilemma.

- Dominant strategy dominates other strategy, if the outcome of it is better for a given player than the outcome of other strategies for every choice of strategies played by other players (e.g., tit-for-tat).