CSE 486 A

Emil Sayahi

11th October 2023

Lecture notes from the 2023 undergraduate course 'Introduction to Artificial Intelligence', given by Professor Khodakhast Bibak at Miami University at Benton Hall in the academic year 2023-2024. This course covers introductory artificial intelligence concepts. Credit for the material in these notes is due to Professor Khodakhast Bibak, while the structure is loosely taken from the in-class lectures. The credit for the typesetting is my own.

Disclaimer: This document will inevitably contain some mistakes—both simple typos and legitimate errors. Keep in mind that these are the notes of an undergraduate student in the process of learning the material, so take what you read with a grain of salt. If you find mistakes and feel like telling me, I will be grateful and happy to hear from you, even for the most trivial of errors. You can reach me by email, in English, at sayahie@miamioh.edu.

For more notes like this, visit my GitHub profile.

Emil Sayahi,
Fall Term: 2023,
Last Update: 11th October 2023,
Miami University

Contents

Wed, 30 August 2023, 11:40am – 1:00pm

Lecture 1: Week 1, Wednesday

1.1 Uninformed Search

Many AI tasks can be formulated as search problems; the goal is to find a *sequence of actions*.

- Puzzles
- Games
- Navigation
- Assignment
- Motion planning
- Scheduling
- Routing

Fri, 1 September 2023, 11:40am – 1:00pm

Lecture 2: Week 1, Friday

Definition 2.1

Uninformed search is a search strategy that uses no problem-specific knowledge. Only the goal test and the successor function are used; the **successor function** generates all possible states. It is not known which non-goal states are better than others. Strategies that know whether one non-goal state is better than another are referred to as **informed search** or **heuristic search** strategies.

There are five major types of uninformed search strategies:

- Breadth-first search (BFS)
- Uniform-cost search (UCS)
- Depth-first search (DFS)
- Depth-limited search (DLS)
- Iterative deepening search (IDS)

All of these uninformed search strategies are distinguished by the *order* in which nodes are expanded.

2.1 Breadth-first search (BFS)

Breadth-first search operates level-by-level, expanding all nodes at a given level before expanding any nodes at the next level. On a given level, nodes are expanded from left to right by convention.

Definition 2.2

Breadth-first search is implemented using a **first in, first out (FIFO) queue**. The FIFO queue is a data structure that supports two operations: **enqueue** and **dequeue**. The enqueue operation adds an element to the end of the queue, and the dequeue operation removes an element from the front of the queue.

Definition 2.3

If a solution exists, breadth-first search will find it in finite time, provided that the branching factor is finite and the depth of the solution is finite; this means that breadth-first search is **complete**. Breadth-first search is not always **optimal**, however, as the solution found may not have the minimum cost. It is optimal when all edges have the same cost, no cost, or when the cost is a non-decreasing function of the depth of the node.

Definition 2.4

The **time complexity** of an algorithm is the number of steps required to solve a problem of size n, where n is the size of the input; in the worst-case of breadth-first search, the goal node would be the very last node explored (ie, every vertex and edge is explored; $O(|V| + |E|)$). The **space complexity** of an algorithm is the maximum amount of memory required to solve a problem of size n; in the worst-case of breadth-first search, the goal node is discovered after all verticies are explored & stored in memory $O(|V|)$.

Definition 2.5

Complexity is expressed in terms of three quantities:

- b is the **branching factor**, or the maximum number of children, or 'successors', of any node.
- d is the **depth** of the shallowest (ie, closest to the root) goal node.
- m is the **maximum length** of any path in the state space.

The time and space complexity of breadth-first search is exponential, $O(b^d)$, where b is the branching factor and d is the depth of the shallowest goal node. This is because the number of nodes expands exponentially with the depth of the tree.

2.2 Uniform-cost search (UCS)

Uniform-cost search expands the node n with the *lowest* path cost $g(n)$ instead of expanding the shallowest node, where $g(n)$ returns the cost of the path from the starting node, s, to the current node, n. This is also referred to as 'Dijkstra's algorithm'. This algorithm uses a priority queue to order nodes in the frontier list by path cost, with the lowest cost node at the front of the queue.

Wed, 6 September 2023, 11:40am – 1:00pm

Lecture 3: Week 2, Wednesday

3.1 Uniform-cost search (UCS) – continued

UCS is optimal and complete, but its time and space complexity remain exponential in the worst case ($O(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor})$), where all edge costs are at least ε, $\varepsilon > 0$, and C^* is the cost of the optimal solution).

3.2 Depth-first search (DFS)

Depth-first search expands the *deepest* node first, removing elements from memory as it proceeds.

Definition 3.6

Depth-first search performs **chronological backtracking**; when a search hits a dead end, it backs up to the level above.

Definition 3.7

DFS is not complete without a **depth bound**, D.

DFS is not optimal or complete. It has an exponential time complexity of $O(b^m)$ and a linear space complexity of $O(bm)$, where M is the maximum length of any path in the state space, and b is the branching factor.

3.3 Iterative deepening search (IDS)

IDS is a combination of BFS and DFS. It performs a DFS with a depth bound, D (typically starting at 1), that increases with each iteration. It is complete (when there are no loops) and optimal, and has a time complexity of $O(b^d)$ and a space complexity of $O(bd)$, where d is the depth of the shallowest goal node.

Definition 3.8

Iterative deepening search is an example of an **'anytime' algoirhtm**; it can return a valid solution to a problem even if it is interrurpted before concluding. It is expected to find better solutions as it continues running.

Generally, IDS is the preferred uninformed search algorithm when the search space is large and the depth of the solution is not known.

Fri, 8 September 2023, 11:40am – 1:00pm

Lecture 4: Week 2, Friday

4.1 Informed Search

Definition 4.9

Informed search algorithms use problem-specific knowledge (ie, **domain knowledge**) to find solutions more efficiently than uninformed search algorithms. They use a **heuristic function** to estimate the cost of the cheapest path from a given node to a goal node. The heuristic function is denoted $h(n)$, where n is a node in the search tree. $h(n) \geq 0$ for all n, while an $h(n)$ close to 0 means that n is close to a goal node, while an $h(n)$ that is very large means that n is far from a goal node.

4.2 A^* Search

A^* search is an informed search algorithm that uses a heuristic function to estimate the cost of the cheapest path from a given node to a goal node. It uses a **cost function**, $f(n)$, to estimate the cost of the cheapest path from the start node to a goal node through n. $f(n)$ is defined as $f(n) = g(n)+h(n)$. $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is the heuristic function. A^* search expands the nodes on the frontier in order of increasing $f(n)$ values (ie, the node with the lowest $f(n)$ is expanded first).

Definition 4.10

A heuristic, h, is **admissible** if it never overestimates the cost of reaching the goal, ie, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost of reaching the goal from n. An admissible heuristic is **optimistic**. The straight-line distance (h_{SLD}) between two points is an admissible heuristic for the problem of finding the shortest path between them. If the heuristic is admissible, then A^* search is optimal.

The time & space complexity of A^* search is polynomial when h satisfies $|h(n) - h^*(n)| = O(\log h^*(n))$. When $h(n) = 0$, this condition is *not* satisfied, and, in effect, A^* search becomes UCS.

Wed, 13 September 2023, 11:40am – 1:00pm

Lecture 5: Week 3, Wednesday

Definition 5.11

Multiagent environments are environments in which multiple agents share the same environment.

Contingency plans are necessary to account for the unpredictability of other agents.

Definition 5.12

Each agent has its own **utility function** that maps states to real numbers.

Definition 5.13

A **zero-sum game** is a game in which the sum of the utilities of all agents is zero. A **game** is a decision-making problem, which is a multiagent environment in which the agents' goals are in conflict. A **competitive game** is a game in which the agents' utility functions are maximised by different states.

Definition 5.14

A **game** is defined by:

- S_0, the initial state.
- PLAYER(s), which returns the player whose turn it is in state s.
- ACTIONS(s), which returns the set of legal moves in state s.
- RESULT(s, a), which returns the state resulting from playing action a in state s.
- TERMINAL-TEST(s), which returns if s is in its terminal state.
- UTILITY(s, p) (referred to as the objective function or payoff function), which returns the final numeric value for a game that ends in terminal state s for a player p.

5.1 α-β pruning

Definition 5.15

Minimax is a decision rule for minimizing the possible loss for a worst case (maximum loss) scenario. It is a recursive algorithm for choosing the next move in an n-player game, usually a two-player game. A value is associated with each position or state of the game. This value is computed by means of a **position evaluation function** and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. If it is A's turn to move, A gives a value to each of their legal moves. A will choose the move with the maximum value of the minimum values resulting from their opponent's possible following moves. If it is B's turn to move, B gives a value to each of their legal moves. B will choose the move with the minimum value of the maximum values resulting from their opponent's possible following moves.

Searching a complete tree takes $O(b^m)$ time, where b is the branching factor and m is the maximum depth of the tree. This is too slow for most games. We can prune the tree to reduce the number of nodes that need to be explored.

Definition 5.16

Pruning is the process of removing parts of a tree that are not relevant to the computation. α-β **pruning** is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Fri, 15 September 2023, 11:40am – 1:00pm

Lecture 6: Week 3, Friday

6.1 α-β pruning – continued

α-β pruning has the property of reducing the branching factor, b, to its square root (ie, $b \overset{\alpha\beta}{\to} \sqrt{b}$). This is because the algorithm is able to prune away half of the branches at each level of the tree.

Wed, 20 September 2023, 11:40am – 1:00pm

Lecture 7: Week 4, Wednesday

7.1 Gradient Descent

In many real-world scenarios, states are continuous variables.

Example. Suppose you have several cities with nearby airports, and you want to build three new airports, while minimising the distance from each city to its nearest airport. You could model this problem as a continuous optimization problem, where C_i denote the cities that have the airport i as their nearest airport. If you define (x_i, y_i) as the coordinates of the airport i, and (x_c, y_c) as the coordinates of the city c. The aim is to minimise the function f, $f(x_1, y_1, x_1, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$. There is no need to place a $\sqrt{}$ in the function around the distance formula, since the square root is a monotonic function, and the minimum of the function f is the same as the minimum of $\sqrt{f}$.

Solution:-

A common approach to solving optimisation problems involves calculating the gradient; $\nabla f = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3})$. We can apply the update rule $x_i \leftarrow x_i - \alpha \frac{\partial f}{\partial x_i}$ (ie, $x \leftarrow x - \alpha \nabla f$) to each of the coordinates of the airports, where α is the learning rate. We can then repeat this process until the gradient is close to zero. In this scenario, $\frac{\partial f}{x_i} = 2 \sum_{c \in C_i} (x_i - x_c)$. This applies to y_i as well. In order to choose α, we can use Newton's method.

Definition 7.17: Newton's method

In 1669, Sir Isaac Newton discovered a method for finding the roots of a function g that consists of iteratively applying the update rule $x \leftarrow x - \frac{g(x)}{g'(x)}$ until $g(x)$ is close to zero. This method is called **Newton's method**. In optimisation, the goal is to find a point where ∇g is 0; the update rule can be rewritten as $x \leftarrow x - \frac{\nabla g(x)}{\nabla^2 g(x)}$, where $\nabla^2 g(x)$ is the Hessian matrix of g at x as g'' is multivariate. Therefore, the update rule becomes $x \leftarrow x - H_g^{-1}(x) \nabla g(x)$.

Definition 7.18: Gradient descent

Gradient descent is an algorithm for finding the minimum of a function f that takes a real number x and returns a real number $f(x)$. The gradient descent algorithm is as follows:

1. Pick a random value for x.
2. Compute the gradient of $f(x)$ at x.
3. Update x by taking a small step in the direction of the negative gradient.
4. Repeat steps 2 and 3 until x converges.

Definition 7.19: Hessian matrix

A **Hessian matrix** of second derivatives is a matrix whose elements are the second partial derivatives of a function; $H_f(x)$ would have its elements, H_{ij}, given by $\frac{\partial^2 f}{\partial x_i \partial x_j}$. The Hessian matrix is used to determine whether a critical point of a function is a local maximum, local minimum, or saddle point.

In the prior airport example of gradient descent, the Hessian matrix, H_f^{-1}, would be:

$$\begin{pmatrix} \frac{1}{2|C_1|} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2|C_2|} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2|C_3|} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2|C_4|} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2|C_5|} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2|C_6|} \end{pmatrix}_{6\times 6}$$

Fri, 6 October 2023, 11:40am – 1:00pm

Lecture 8: Week 6, Friday

8.1 Constraint Satisfaction Problems (CSPs)

A constraint satisfaction problem consists of three components:

- A set of variables $X_1, X_2, \ldots, X_n$.
- A set of domains $D_1, D_2, \ldots, D_n$, where D_i is the domain of the variable X_i—that is, $X_i \in D_i$.
- A set of constraints that specify allowable combinations of values.

Definition 8.20

To solve a CSP, the state space must be defined:

- **State**: assignment of values to some or all variables.
- **Consistent assignment**: assignment that does not violate any constraints.
- **Partial assignment**: assignment that does not assign values to all variables.
- **Complete assignment**: assignment that assigns values to all variables.
- **Solution**: an assignment which is consistent & complete.

Definition 8.21

The **four colour theorem** (Appel and Haken, 1976) states that any map can be coloured using only four colours, so that no two adjacent regions have the same colour.

Example. If we wished to colour a map of Australia, we could represent the problem as a CSP with the following variables and domains:

- $X = \{\text{WA}, \text{NT}, \text{Q}, \text{NSW}, \text{NSW}, \text{V}, \text{SA}, \text{T}\}$
- $D_i = \{\text{Red}, \text{Green}, \text{Blue}\}$
- $C = \{\text{SA} \neq \text{WA}, \text{SA} \neq \text{NT}, \text{SA} \neq \text{Q}, \text{SA} \neq \text{NSW}, \text{SA} \neq \text{V}, \text{WA} \neq \text{NT}, \text{NT} \neq \text{Q}, \text{Q} \neq \text{NSW}, \text{NSW} \neq \text{V}\}$

CSPs can be more efficient than state space searchers, as constraints can eliminate large portions of the search space. In this case, setting $\text{SA} = \{\text{Blue}\}$ reduces the number of assignments from 3^5 to 2^5—from 243 to 32.

Example. If we wished to schedule the steps of a car assembly line, we could represent the problem as a CSP. The assembly line performs the following steps:

1. Installing the axles, Axle_F & Axle_B, taking 10 minutes each.
2. Affix the wheels, Wheel_{RF}; Wheel_{LF}; Wheel_{RB}; and Wheel_{LB}, taking 1 minute each.
3. Tighten the nuts for each wheel, Nut_{RF}; Nut_{LF}; Nut_{RB}; and Nut_{LB}, taking 2 minutes each.
4. Affix the hubcaps, Cap_{RF}; Cap_{LF}; Cap_{RB}; and Cap_{LB}, taking 1 minute each.
5. Inspect the final assembly, Inspect, which takes 3 minutes.

The problem is finding when each task should be started in the time interval of $[0, 30]$ minutes. We can represent the problem as a CSP with the following variables and domains:

$$\begin{aligned} X = \{&\text{Axle}_F, \text{Axle}_B, \text{Wheel}_{RF}, \text{Wheel}_{LF}, \text{Wheel}_{RB}, \text{Wheel}_{LB}, \\ &\text{Nut}_{RF}, \text{Nut}_{LF}, \text{Nut}_{RB}, \text{Nut}_{LB}, \\ &\text{Cap}_{RF}, \text{Cap}_{LF}, \text{Cap}_{RB}, \text{Cap}_{LB}, \\ &\text{Inspect}\} \\ D_i = [0, 27] \end{aligned}$$

We have the following precedence constraints:

$$\begin{aligned} \text{Axle}_F + 10 \leq \text{Wheel}_{RF}; &\text{Axle}_F + 10 \leq \text{Wheel}_{LF}; \\ \text{Axle}_B + 10 \leq \text{Wheel}_{RB}; &\text{Axle}_B + 10 \leq \text{Wheel}_{LB}; \\ \text{Wheel}_{RF} + 1 \leq \text{Nut}_{RF}; &\text{Nut}_{RF} + 2 \leq \text{Cap}_{RF}; \\ \text{Wheel}_{LF} + 1 \leq \text{Nut}_{LF}; &\text{Nut}_{LF} + 2 \leq \text{Cap}_{LF}; \\ \text{Wheel}_{RB} + 1 \leq \text{Nut}_{RB}; &\text{Nut}_{RB} + 2 \leq \text{Cap}_{RB}; \\ \text{Wheel}_{LB} + 1 \leq \text{Nut}_{LB}; &\text{Nut}_{LB} + 2 \leq \text{Cap}_{LB}. \end{aligned}$$

Additionally, we have the constraint that $X_i + T_i \leq \text{Inspect}$ for each X_i, where T_i is the duration of task X_i. Therefore, the solution to this problem would be an assignment of each variable to a value in D_i such that every constraint is satisfied.

Considering the above example, if we have four workers to install the wheels, but only one pair of workers can install an axle at a time (ie, Axle_F and Axle_B cannot coincide), then we have a **disjunctive constraint**, $(\text{Axle}_F + 10 \leq \text{Axle}_B) \vee (\text{Axle}_B + 10 \leq \text{Axle}_F)$.

Definition 8.22

A **linear program** is an optimisation problem where the objective function and constraints are all linear.

Example. Given the previous example, we could have a linear program that aimed to minimise Inspect such that $\forall X_i \in X - \{\text{Inspect}\} : X_i + T_i \leq \text{Inspect}$;

$$\begin{gathered}
\text{Axle}_F + 10 \leq \text{Wheel}_{RF}; \text{Axle}_F + 10 \leq \text{Wheel}_{LF}; \\
\text{Axle}_B + 10 \leq \text{Wheel}_{RB}; \text{Axle}_B + 10 \leq \text{Wheel}_{LB}; \\
\text{Wheel}_{RF} + 1 \leq \text{Nut}_{RF}; \text{Nut}_{RF} + 2 \leq \text{Cap}_{RF}; \\
\text{Wheel}_{LF} + 1 \leq \text{Nut}_{LF}; \text{Nut}_{LF} + 2 \leq \text{Cap}_{LF}; \\
\text{Wheel}_{RB} + 1 \leq \text{Nut}_{RB}; \text{Nut}_{RB} + 2 \leq \text{Cap}_{RB}; \\
\text{Wheel}_{LB} + 1 \leq \text{Nut}_{LB}; \text{Nut}_{LB} + 2 \leq \text{Cap}_{LB}.
\end{gathered}$$

While linear programming is appropriate for optimisation problems, CSPs are not necessarily optimisation problems; we're only seeking some solution that satisfies all of the given constraints. For example, the four colour theorem is not an optimisation problem, as there is no objective function to minimise or maximise. CSPs can have non-linear constraints, such as complex constraints on discrete variables (such as in the eight queens puzzle), or constraints on continuous variables (such as in the travelling salesman problem), while linear programs must only have linear constraints.

Notes