4 BEYOND CLASSICAL SEARCH

- Above we studied systematic search that maintains paths in memory.
- In many problems, however, the path to the goal is irrelevant: 8-queens, integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, vehicle routing, …
- It suffices to know the solution to the problem.
- Local search algorithms operate using a single current state and generally move to neighbors of that state.
- Typically, the paths followed by the search are not retained.
- They use very little memory, usually a constant amount.
- Local search algorithms lead to reasonable results in large or infinite (continuous) state spaces for which systematic search methods are unsuitable.

4.1 Local Search Algorithms and Optimization Problems

- Local search algorithms are also useful for solving pure optimization problems.
- In optimization the aim is to find the best state according to an objective function.
- Optimization problems are not always search problems in the same sense as they were considered above.
  - For instance, Darwinian evolution tries to optimize reproductive fitness.
  - There does not exist any final goal state (goal test).
  - Neither does the cost of the path matter in this task.
Optimization of the value of objective function can be visualized as a state space landscape, where the height of peaks and depth of valleys corresponds to the value of the function. A search algorithm giving the optimal solution to a maximization problem comes up with the global maximum. Local maxima are higher peaks than any of their neighbors, but lower than the global maximum.

4.1.1 Hill climbing search

- In hill-climbing one always chooses a successor of the current state $s$ that has the highest value for the objective function $f$.

\[ \max_{s' \in S(s)} f(s') \]

- Search terminates when all neighbors of the state have a lower value for the objective function than the current state has.
- Most often search terminates in a local maximum, sometimes by chance, in a global maximum.
- Also plateaux cause problems to this greedy local search.
- On the other hand, improvement starting from the initial state is often very fast.
• Sideways moves can be allowed when search may proceed to states that are as good as the current one
• *Stochastic* hill-climbing chooses at random one of the neighbors that improve the situation
• Neighbors can, for example, be examined in random order and choose the first one that is better than the current state
• Also these versions of hill-climbing are incomplete because they can still get stuck in a local maximum
• By using *random restarts* one can guarantee the completeness of the method
  – Hill-climbing starts from a random initial state until a solution is found

### 4.1.2 Simulated annealing

• A *random walk* — moving to a successor chosen uniformly at random from the set of successors independent of whether it is better than the current state — is a complete search algorithm, but when unsupervised also extremely inefficient
• Let us allow “bad” moves with some probability $p$
• The probability of transitions leading to worse situation decreases exponentially with time (“temperature”)
• We choose a candidate for transition randomly and accept it if
  1. the objective value improves;
  2. otherwise with probability $p$
• If temperature is lowered slowly enough, this method converges to a global optimum with probability $\rightarrow 1$
4.1.3 Local beam search

- The search begins with $k$ randomly generated states
- At each step, all the successors of all $k$ states are generated
- If any one of the successors is a goal, the algorithm halts
- Otherwise, it selects the $k$ best successors from the complete list and repeats
- The parallel search of beam search leads quickly to abandoning unfruitful searches and moves its resources to where the most progress is being made
- In stochastic beam search the maintained successor states are chosen with a probability based on their goodness

4.1.4 Genetic algorithms

- Genetic algorithms (GA) apply to search operations familiar from evolution and inheritance
- A GA, like beam search, starts from a population of $k$ randomly generated states
- The states are represented as strings and they are now called individuals
- To come up with the population of the next generation, all individuals are rated with a fitness function
- The probability of an individual to reproduce depends on its fitness (selection)
- The genetic operator crossover is applied to random pairs of selected individuals so that a suitable cut point is chosen from both strings and their suffixes are swapped
Finally, each location in the child strings is subject to possible mutation. In mutation characters are replaced with other with a (very) small independent probability. The success of GAs usually requires carefully chosen coding of individuals and restricting genetic operations so that the children make sense as solutions. Nowadays very popular heuristic search method, but quite inefficient.

4.2 Local Search in Continuous Spaces

Let the objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ be a function on six continuous-valued variables. The gradient of the objective function $\nabla f$ is a vector that gives the magnitude and direction of the steepest slope:

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

In many cases, we cannot solve equation $\nabla f = 0$ in closed form (globally), but can compute the gradient locally. We can perform steepest-ascent hill climbing by updating the current state $x$ via the formula

$$x \leftarrow x + \alpha f(x),$$

where $\alpha$ is the step size, a small constant.
If the objective function is not differentiable, the empirical gradient can be determined by evaluating the response to small increments and decrements in each coordinate. Adjusting the value of constant $\alpha$ is central: if $\alpha$ is too small, too many steps are needed; if $\alpha$ is too large, the search could overshoot the maximum. Line search repeatedly doubles the value of $\alpha$ until $f$ starts to decrease again. Equations of the form $g(x) = 0$ can be solved using the Newton-Raphson method. It works by computing a new estimate for the root $x$ according to Newton’s formula:

$$x \leftarrow x - g(x)/g'(x)$$

To find a maximum or minimum of $f$, we need to find $x$ s.t. the gradient is zero; i.e., $\nabla f(x) = 0$. Setting $g(x) = \nabla f$ in Newton’s formula and writing it matrix-vector form, we have $x \leftarrow x - H_f^{-1}(x) \nabla f(x)$, where $H_f(x)$ is the Hessian matrix of second derivatives.

The Hessian has quadratic number of entries, and Newton-Raphson becomes expensive in high-dimensional spaces. Local search suffers from local maxima, ridges, and plateaus in continuous state spaces just as much as in discrete spaces. Constrained optimization requires a solution to satisfy some hard constraints on the values of each variable. E.g., in linear programming the constraints must be linear inequalities forming a convex region and objective function is also linear.
4.5 Online Search and Unknown Environments

- An online search agent has to react to its observations immediately without contemplating on far-reaching plans.
- In an unknown environment exploration is necessary: the agent needs to experiment on its actions to learn about their consequences to learn about the states of the world.
- Now an agent cannot compute the successors of the current state, but has to explore what state follows from an action.
- It is common to contrast the cost of the path followed by an online algorithm to the cost of the path followed by an offline algorithm.
- The ratio of these costs is called the competitive ratio of the online algorithm.
To determine the competitive ratio of an online algorithm, we compare the cost of the path followed by it to the cost of the path followed by the agent if it knew the search space in advance.

- The smaller the competitive ratio, the better.

- Online algorithms can be analyzed by considering their performance as a game with a malicious adversary.
- Oblivious adversaries are not as interesting.
- The adversary gets to choose the state space on the fly while the agent explores it.
- The adversary’s intention is to force the online algorithm to perform poorly.
Not all of the offline search algorithms that we have considered are suitable for online search:

- For example, A* is essentially based on the fact that one can expand any node generated to the search tree.
- An online algorithm can expand only a node that it physically occupies.
- Depth-first search only uses local information, except when backtracking.
- Hence, it is usable in online search (if actions can physically be undone).
- Depth-first search is not competitive: one cannot bound the competitive ratio.

Hill-climbing search already is an online algorithm, but it gets stuck at local maxima:

- Random restarts cannot be used:
  - the agent cannot transport itself to a new state.
- Random walks are too inefficient.
- Using extra space may make hill-climbing useful in online search.
- We store for each state $s$ visited our current best estimate $H(s)$ of the cost to reach the goal.
- Rather than staying where it is, the agent follows what seems to be the best path to the goal based on the current cost estimates for its neighbors.
- At the same time the value of a local minimum gets flattened out and can be escaped.
5 ADVERSARIAL SEARCH

- Let us turn away from searching a path from the initial state to a goal state and consider competitive environments instead.
- There is an adversary that may also make state transitions and the adversary wants to throw our good path off the rails.
- The aim in adversarial search is to find a move strategy that leads to a goal state independent of the moves of the adversary.
- Two-player deterministic, turn-taking, two-player, zero sum (board) games of perfect information.
- In the end of the game one of the players has won and the other one has lost.
5.2 Optimal Decisions in Games

- Let the two players be called \textbf{min} and \textbf{max}
- In the initial state the board position is like the rules of the game dictate and the player \textbf{max} is the first to move
- The successor function determines the legal moves and resulting states
- A terminal test determines when the game is over
- A \textit{utility function} (or the payoff \textit{f.}) gives a numeric value for the terminal states, in chess the value may be simply 0, ½, 1 or, e.g., the sum of the pieces remaining on the board
- \textbf{max} aims at maximizing and \textbf{min} aims at minimizing the value of the utility function
- The initial state and the successor function determine a \textit{game tree}, where the players take turns to choose an edge to travel

In our quest for the optimal game strategy, we will assume that also the adversary is infallible

- Player \textbf{min} chooses the moves that are best for it
- To determine the optimal strategy, we compute for each node \( n \) its \textit{minimax} value:

\[
\text{MM}(n) = \begin{cases} 
\text{Utility}(n), & \text{if } n \text{ is a terminal state} \\
\max_{s \in S(n)} \text{MM}(s), & \text{if } n \text{ is a max node} \\
\min_{s \in S(n)} \text{MM}(s), & \text{if } n \text{ is a min node}
\end{cases}
\]
**function** MINMAX-DECISION\((state)\) **returns** an action

1. return \(\text{arg}\max_{a\in\text{ACTIONS}(state)} \text{MIN-VALUE}(\text{RESULT}(s, a))\)

**function** MAX-VALUE\((state)\) **returns** a utility value

1. if TERMINAL-TEST\((state)\) then return UTILITY\((state)\)
2. \(v \leftarrow -\infty\)
3. for each \(a\) in ACTIONS\((state)\) do
4. \(v \leftarrow \max(v, \text{MIN-VALUE}(\text{RESULT}(state, a)))\)
5. return \(v\)

**function** MIN-VALUE\((state)\) **returns** a utility value

1. if TERMINAL-TEST\((state)\) then return UTILITY\((state)\)
2. \(v \leftarrow \infty\)
3. for each \(a\) in ACTIONS\((state)\) do
4. \(v \leftarrow \min(v, \text{MAX-VALUE}(\text{RESULT}(state, a)))\)
5. return \(v\)
The play between two optimally playing players is completely determined by the minimax values.

For \textit{max} the minimax values give the worst-case outcome — the opponent \textit{min} is optimal.

If the opponent does not choose the best moves, then \textit{max} will do at least as well as against \textit{min}.

There may be other strategies against suboptimal opponents that do better than the minimax strategy.

The minimax algorithm performs a complete depth-first exploration of the game tree, and therefore the time complexity is $O(b^m)$, where $b$ is the number of legal moves at each point and $m$ is the maximum depth.

For real games, exponential time cost is totally impractical.

5.3 Alpha-Beta Pruning

The exponential complexity of minimax search can be alleviated by pruning the nodes of the game tree that get evaluated.

It is possible to compute the correct minimax decision without looking at every node in the game tree.

For instance, to determine the minimax value of the game tree above, two leaves can be left unexplored, because

\[
\text{MM(root)} = \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\
= \max(3, \min(2, x, y), 2) \\
= \max(3, z, 2), \text{where } z \leq 2 \\
= 3
\]
• The value of the root is independent the values of leaves $x$ and $y$
• The general principle of pruning is:
  – In considering a move to a node $n$ anywhere in the tree,
  – If the player has a better choice $m$ either at the parent
    node of $n$ or at any choice point further up,
  – then $n$ will never be reached in actual play

• Alpha-beta pruning gets its name from the parameters that
describe bounds on the backed-up values that appear
anywhere along the path
  – $\alpha$ = the value of the best (highest-value) choice we have
    found so far at any choice point along the path for $\text{max}$
  – $\beta$ = the value of the best (lowest-value) choice we have
    found so far at any choice point along the path for $\text{min}$

• Alpha-beta search updates the values of $\alpha$ and $\beta$ as it goes
  along
• As soon as the value of the current node is known to be
  worse than the current $\alpha$ (max) or $\beta$ (min) the remaining
  branches can be pruned
• The effectiveness of alpha-beta pruning is highly dependent
  on the order in which the successors are examined
• In the previous example, we could not prune any of the
  successors of the last branch because the worst successors
  (from the point of view of $\text{min}$) were generated first
• If the third successor had been generated first, we would have
  been able to prune the other two
function ALPHA-BETA-SEARCH(state)
    returns an action
1. $v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty)$
2. return the action in ACTIONS(state) with value $v$

function MAX-VALUE(state, $\alpha$, $\beta$)
    returns a utility value
1. if TERMINAL-TEST(state) then return UTILITY(state)
2. $v \leftarrow -\infty$
3. for each $a$ in ACTIONS(state) do
4.     $v \leftarrow \max(v, \text{MIN-VALUE}(\text{RESULT}(state, a, \alpha, \beta)))$
5.     if $v \geq \beta$ then return $v$
6.     $\alpha \leftarrow \max(\alpha, v)$
7. return $v$

function MIN-VALUE(state, $\alpha$, $\beta$)
    returns a utility value
1. if TERMINAL-TEST(state) then return UTILITY(state)
2. $v \leftarrow +\infty$
3. for each $a$ in ACTIONS(state) do
4.     $v \leftarrow \min(v, \text{MAX-VALUE}(\text{RESULT}(state, a, \alpha, \beta)))$
5.     if $v \leq \alpha$ then return $v$
6.     $\beta \leftarrow \min(\beta, v)$
7. return $v$
If the best successors could be examined first, then alpha-beta needs to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax.

- The effective branching factor becomes $\sqrt{B}$ instead of $b$.
- For example in chess, this would in practice mean factor of 6 instead of the original 35.
- In games one cannot evaluate full game trees, but one rather aims at evaluating partial game trees as many moves ahead as possible (two half-moves = ply).
- In other words, Alpha-beta could look ahead roughly twice as far as minimax in the same amount of time.

### Deep Blue

- A chess-playing parallel computer developed at IBM, which in 1997 beat the world champion Garry Kasparov in a six-game exhibition match.
- Searched on average 126 million nodes per second (peak 330 million nodes).
- Routine search depth: 14.
- Standard iterative-deepening $\alpha$-$\beta$ search.
- Key 4 success: ability to generate extension beyond the depth limit for sufficiently interesting lines of moves (up to 40 plies).
- Evaluation function had over 8000 features.
- Large “opening book” and endgame library.
5.4 Imperfect, Real-Time Decisions

- Searching through the whole (pruned) game tree is too inefficient for any realistic game
- Moves must be made in a reasonable amount of time
- One has to cut off the generation of the game tree to some depth and the absolute terminal node values are replaced by heuristic estimates
- Game positions are rated according to how good they appear to be (with respect to reaching a goal state)
- A basic requirement for a heuristic evaluation function is that it orders the terminal states in the same way as the true utility function
- Of course, evaluation of game positions may not be too inefficient and the evaluation function should be strongly correlated with the actual chances of winning

Most evaluation functions work by calculating features of the state
- E.g., in chess the number of pawns possessed by each side could be one feature
- As game positions are mapped to the values of the chosen features, different states may look equivalent, even though some of them lead to wins, some to draws, and some to losses
- For such an equivalence class of states, we can compute the expected end result
- If, e.g., 72% of the states encountered in the category lead to a win (utility $+1$), 20% to a loss ($0$) and 8% to a draw ($\frac{1}{2}$), then the expected value of a game continuing from this category is:

$$\left(0.72 \times 1\right) + \left(0.20 \times 0\right) + \left(0.08 \times \frac{1}{2}\right) = 0.76$$
Because the number of features and their possible values is usually high, the method based on categories is only rarely usable.

Instead, most evaluation functions compute separate numerical contribution for each feature \( f_i \) on position \( s \) and combine them by taking their weighted linear function as the evaluation function:

\[
eval(s) = \sum_{i=1}^{n} w_i f_i(s)
\]

For instance, in chess features \( f_i \) could be the numbers of pawns, bishops, rooks, and queens.

The weights \( w_i \) for these features, on the other hand, would be the material values of the pieces (1, 3, 5, and 9).

Adding up the values of the features involves the strong assumption about the **independence** of the features.

However, e.g., in chess bishops are more powerful in the endgame, when they have a lot of space to maneuver.

For this reason, current programs for chess and other games also use **nonlinear** combinations.

For example, a pair of bishops might be worth slightly more than twice the value of a single bishop, and a bishop is worth more in the endgame than in the beginning.

If different features and weights do not have centuries of experience behind them like in chess, the weights of the evaluation function can be estimated by machine learning techniques.
5.5 Stochastic Games

- Games can include an explicit random element, e.g., by throwing a dice
- A board game with such an element is backgammon
- Although the player knows what her own legal moves are, she does not know what the opponent is going to roll and thus does not know what the opponent's legal moves will be
- Hence, a standard game tree cannot be constructed
- In addition to max and min nodes one must add chance nodes into the game tree
- The branches leading from each chance node denote the possible dice rolls, and each is labeled with the roll and the chance that it will occur
In backgammon one rolls two dice, so there are \(6 + 15\) distinct pairs and their chances of coming up are \(1/36\) and \(1/18\).

Instead of definite minimax values, we can only calculate the expected value, where the expectation is taken over all the possible dice rolls that could occur.

E.g., the expected value of a \(\text{max}\) node \(n\) is now determined as:

\[
\max_{s \in S(n)} E[M(s)]
\]

In a chance node \(n\) we compute the average of all successors weighted by their probability \(P(s)\) (the required dice roll occurs):

\[
\sum_{s \in S(n)} P(s) \cdot E[M(s)]
\]

Evaluating positions in a stochastic game is a more delicate matter than in a deterministic game.

Including the element of chance increases the time complexity of game tree evaluation to \(O(b^m n^m)\), where \(n\) is the number of distinct dice rolls.

In backgammon \(n = 21\) and \(b\) is usually around \(20\), but in some situations can be as high as \(4000\) for dice rolls that are doubles.

Even if the search depth is limited, the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far for most stochastic games.

Alpha-beta pruning concentrates on likely occurrences.

In a game with dice, there are no likely sequences of moves.

However, if there is a bound on the possible values of the utility function, one can prune a game tree including chance nodes.