Heuristic $h_2$ gives a stricter estimate than $h_1$: $h_2(n) \leq h_1(n)$

- We say that the former dominates the latter
- $A^*$ expands every node for which $f(n) < C^* \Leftrightarrow h(n) < C^* - g(n)$
- Hence, a stricter heuristic estimate directly leads to more efficient search

- The branching factor of 8-puzzle is about 3
- Effective branching factor measures the average number of nodes generated by $A^*$ in solving a problem
- For example, when $A^*$ applies heuristic $h_1$, the effective factor in 8-puzzle is on average circa 1.4, and using heuristic $h_2$ c. 1.25

3.6.2 Generating admissible heuristics from relaxed problems

- To come up with heuristic functions one can study relaxed problems from which some restrictions of the original problem have been removed
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem (does not over-estimate)
- The optimal solution in the original problem is, by definition, also a solution in the relaxed problem
- E.g., heuristic $h_1$ for the 8-puzzle gives perfectly accurate path length for a simplified version of the puzzle, where a tile can move anywhere
- Similarly $h_2$ gives an optimal solution to a relaxed 8-puzzle, where tiles can move also to occupied squares
If a collection of admissible heuristics is available for a problem, and none of them dominates any of the others, we can use the composite function
\[ h(n) = \max\{ h_1(n), \ldots, h_m(n) \} \]

The composite function dominates all of its component functions and is consistent if none of the components overestimates.

One way of relaxing problems is to study subproblems.

E.g., in 8-puzzle we could study only four tiles at a time and let the other tiles wander to any position.

By combining the heuristics concerning distinct tiles into one composite function yields a heuristic function, that is much more efficient than the Manhattan distance.

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 \textit{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 \textit{global maximum}.
- \textit{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 $f$
  
  $$\max_{s' \in \text{neigh}(s)} f(s')$$

- 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
  \[
  \nabla f = \begin{pmatrix}
  \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial y_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial y_2} & \frac{\partial f}{\partial x_3} & \frac{\partial f}{\partial y_3}
  \end{pmatrix}
  \]
- 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 \( \hat{z} \) via the formula
  \[
  \hat{z} \leftarrow \hat{z} + \alpha \nabla f(\hat{z}),
  \]
  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
  \[
  x \leftarrow x - \frac{g(x)}{g'(x)}
  \]
To find a maximum or minimum of \( f \), we need to find \( \mathbf{z} \) s.t. the gradient is zero; i.e., \( \nabla f(\mathbf{z}) = \mathbf{0} \).

Setting \( g(\mathbf{z}) = \nabla f(\mathbf{z}) \) in Newton’s formula and writing it matrix-vector form, we have
\[
\mathbf{z} \leftarrow \mathbf{z} - H_f^{-1}(\mathbf{z}) \nabla f(\mathbf{z}),
\]
where \( H_f(\mathbf{z}) \) is the Hessian matrix of second derivatives,
\[
H_f = \frac{\partial^2 f}{\partial x_i \partial x_j}.
\]

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 *min* and *max*.
- In the initial state the board position is like the rules of the game dictate and the player *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 *utility function* (or the payoff f.) gives a numeric value for the terminal states, in chess the value may be simply 0, \( \frac{1}{2}, 1 \) or, e.g., the sum of the pieces remaining on the board.
- *max* aims at maximizing and *min* aims at minimizing the value of the utility function.
- The initial state and the successor function determine a 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 \textit{min} chooses the moves that are best for it
• To determine the optimal strategy, we compute for each node \( n \) its \textit{minimax} value:

\[
MM(n) = \begin{cases} 
\text{Utility}(n), & \text{if } n \text{ is a terminal state} \\
\max_{v \in \mathcal{V}(n)} MM(v), & \text{if } n \text{ is a max node} \\
\min_{v \in \mathcal{V}(n)} MM(v), & \text{if } n \text{ is a min node} 
\end{cases}
\]
• The play between two optimally playing players is completely determined by the minimax values
• For max the minimax values gives the worst-case outcome — the opponent min is optimal
• If the opponent does not choose the best moves, then max will do at least as well as against 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

\[
\text{\textit{MM(root)}} = \max( \min(3, 12, 8), \min(2, x, y), \min(14, 5, 2) ) = \max( 3, \min(2, x, y), 2 ) = \max( 3, z, 2 ), \text{where } z \leq 2 = 3
\]

- 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:
  - \(a = \) the value of the best (highest-value) choice we have found so far at any choice point along the path for \(\text{max}\)
  - \(b = \) 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 $a$ and $\beta$ as it goes along
• As soon as the value of the current node is known to be worse than the current $a$ ($\max$) 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 $\min$) 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
  $v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty)$
  return the action in ACTIONS(state) with value $v$

function MAX-VALUE(state, $\max$, $\min$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for each $a$ in ACTIONS(state) do
    $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s,a), \alpha, \beta))$
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow \text{MAX}(\alpha, v)$
  return $v$

function MIN-VALUE(state, $\max$, $\min$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow +\infty$
  for each $a$ in ACTIONS(state) do
    $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s,a), \alpha, \beta))$
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow \text{MIN}(\beta, v)$
  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 to the success was the 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