**Game of Go rules:**

Repetition (basic ko): After white captured, black cannot re-captured right away. Black must play somewhere else. Now white has a chance to connect. If white also plays elsewhere, then black can capture.

Legal moves: any empty intersection except point forbidden by repetition and suicide.

End of game and scoring: game end after two successive passes. Score = stones + empty points surrounded by stones (+ komi).

Simple eye: **1.** Single empty point p; **2.** All neighbor points nb(p) occupied by stones of the same color; **3.** All these stones are connected in a single block. Corner, edge: need all diagonal points to connect (1 in corner, 2 on edge). Center: need at least 3 of 4 diagonal points to connect.

**Tromp-Taylor Rules:**

A point P, not colored C, is said to reach C, if there is a path of (vertically or horizontally) adjacent points of P’s color from P to a point of color C.

Clearing a color is the process of emptying all points of that color that don’t reach empty.

A turn is either a pass; or a move that does not repeat an earlier grid coloring.

A move consists of coloring an empty point one’s own color; then clearing the opponent color, and then clearing one’s own color.

**Checking suicide and checking capture**: the main difference between them is that the move can connect several blocks, and none of them may have another liberty.

Undo: for search, we need to consider many alternative moves, so we need undo, especially in dealing with captured stones. Use stack to store each board.

**Floodfill algorithm:**

Keep track of points already visited (e.g. mark them)

Visit all neighbors

DFS

Different ways to implement (explicit recursion e.g. store points to be processed in a stack)

**Heuristic and decision making:**

Heuristic: from Greek word “find” or “discover”; any practical problem solving method; “mental shortcut”, rule of thumb, educated guess, common sense, a rough model, using a similar case for guidance, …

Contrast heuristic: exact solution, exhaustive analysis, precise theory.

Heuristic decision-making and exact methods are both used in computer programs.

Heuristics to make better decisions (**Polya**)

1. Understand the problem: What it the input? What is unknown? What is the condition? Draw a figure. Introduce suitable notation. Draw or write down the important concepts in the problem and their relations. Separate the various parts of the condition. Find smaller parts, functions that make up the required solution
2. Devise a plan: Find connection between data and unknown (function). Do you know a related problem? Can you re-use the previous solution? Could you restate the problem? If you cannot solve the proposed problem, try to solve first a special case / a concrete example. Did you use all the data?
3. Carry out the plan: Write functions to implement your program, test each one separately. Correctness: use assertions to check input and output. Use formal proof (post-condition and loop invariants) for tricky codes
4. Looking back – examine the solution, review/extend. Can you use the result, or the method for some other problem? Refactor code, simplify functions. Extend or generalize functions for other problems. Organize into modules.
5. Example: auxiliary problem (helper function). Decomposing and recombining: separate tree search algorithm from details what to do in each node; floodfill – separate scan of full board from what to do in each area.

Bounded rationality (**Herb Simon**)

Concepts of satisfying.

1. Decision making is not always an optimization problem because decision may involve many factors that are hard to compare. We always cannot make perfect decision because human / computers have limited memory, limited time, limited power of logic, ….
2. Decision makers can satisfice either by finding optimum solution for a simplified world (model), or by finding satisfactory solutions for a more realistic world (direct observation). Humans use heuristics as shortcuts. In games, we have a perfect model, but that is the exception, not true in real world.
3. Normative / descriptive decision theory: what is / how to make a best action.

**Classical game theory: eg. Von Neumann and Morgenstern**

1. Selfish player, try to maximize their money
2. Simplest case: two player zero sum games – my win is your loss
3. Actions can involve random outcomes, but with known probabilities
4. Goal: maximize expected value

Utility function and risk (**Kahneman and Tversky**)

Mathematical models: rewards and utility

1. Utility – measures satisfaction of a consumer with a good. It determines the price that a consumer is willing to pay. Our utility of money does not always scale linearly with the amount of money (risk-averse: slower; risk-neutral; risk-prone: faster).

Example: scaling-up fold or bid? St. Petersburg paradox by Nicolas Bernoulli (the bank one)

1. Maximum expected utility (MEU) – chose action which maximizes your expected utility (von Neumann / Morgenstern)

With uncertain events, but known probabilities, we can compute expected utilities just as we computed expected value. Just replace the values with the utilities in the formula.

1. Marginal utility: increase in consumer satisfaction from having one unit more of a good.

Example: the value of $100 more – high if you are broke / low if you are Bill Gates

1. Humans have systematic cognitive biases. Most are averse to risk and ambiguity. People tend to “anchor” on first impressions. People focus more on changes in their utility than on absolute utilities.

**A simple setting for decision-making (assumption):** **1.** The state of the world, terminal states, possible actions for each state are completely known at each time. **2.** An action changes the state to a new state in a known deterministic way. **3.** No change element (cards drawn).

**Terminology:**

1. History-only state works in principle because *rules plus complete history determine the state exactly*. However, it may be inefficient if we always have to replay all actions from the beginning in order to find the current state.

2. A state space is represented as directed graph.

3. Tree: every action leads to a new state; only one path from root to a node; Size of state space: . Advantage: simplest; single path to a node; no dependency. Disadvantage: duplication; large state space; inefficient.

4. DAG: different move order can lead to same result; Go with full repetition rules. Branching factor at depth d:. Size of state space: 1.

Some practical examples:

Go, TicTacToe: (since one less empty square with each move). However, we ignore illegal move rules (captures, kos) and games that end earlier. Checkers: complicated . Many pieces are blocked at the beginning. When pieces are unblocked, b increases. When pieces get captured, b decreases. When checkers become kings, b strongly increases. Length of game d varies wildly.

Chess: complicated . Length of game d varies wildly.

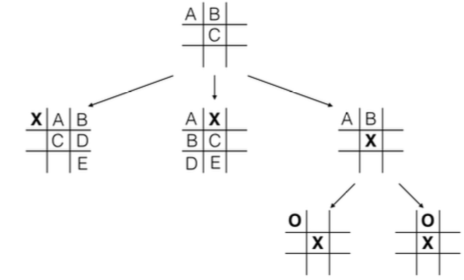
Shogi: . Captured opponent pieces can be reused for yourself in a future move, so b increases with captures. b can be several hundred in endgame with many captured pieces available for “dropping” back on board.

Main problem: need memory to store and recognize equivalent states. Some algorithm designed only for trees, not for DAG.

Limitation: states with different history cannot merge into one node, not equivalent. Board looks the same but different (simple ko).

5. DCG: Go without repetition rules/with simple ko rules.

**Symmetry:** Take symmetry into account (example in TicTacToe): however, most symmetries broken after few moves (good at depth 1 or 2), and almost no reduction deeper in the tree. Reduction of whole state space is limited to some constant factor (less than 8 in Go).



**Improvement and optimization in Go:**

Positional superko(PSK): Ignore whose turn it is, only compare board.

Situational superko(SSK): Compare whose turn it is and board

(use hash table to detect potential repetition)

**Knuth**: premature optimization is the root of all evil

**80-20 rule**: 80% of the improvement can come from 20% of the code.

**Amdahl’s Law**: p = % of program that is speeded up; s = speedup for that part; runtime before optimization: 1; runtime after optimization: (1 – p) + p/s; speed up limit for the whole program: .

**Profiling (build-in cProfile)**: Define a test that runs your program with a typical workload; Run it with a special program called profiler; Profiler tells you details of the program execution. Profiler can be on the function level or instruction level. (How often was a piece of code executed? How long did it take?). Best to avoid calling a function. Second best to speed up a function and avoid unneeded computation.

**Blind search:** we can expect to visit about half of all states before hitting the goal, no matter which blind search we use and no matter which type of graph of the state space is (tree, DAG, DCG, …).

. It takes i + 1 total visits to reach . Expected number of visits to reach goal: .

**Heuristic search: 1.** Search: systematically look ahead into the future. **2.** Heuristic evaluation. **3.** Simulation (one of exploration): look ahead into the future by sampling sequences of future states. **4.** Goals: use heuristics when it works AND have a strong method working even when heuristics fails.

**Heuristic as payoff in win/loss games:** h(s) = -1 if s is a sure loss for the player; h(s) = 1 is s is a sure win for the player; -1 < h(s) < 1 for all other s; draw: h(s) = 0; payoff for opponent: – h(s); translation between winning probability and payoff: Winning probability – p, Payoff – v, Translate by v = 2p – 1 (Linear mapping).

**Heuristic as scores in win/loss games:** h(s) = true score if s is a terminal position; payoff for opponent: -h(s) (zero sum)

**Proof tree/solution tree (a winning strategy):** Subtree P of game tree G is a proof tree iff: P contains the root of G; All leaf nodes of P are wins; If interior AND node is in P, then all its children are in P; If interior OR node is on P, at least one child is in P.

Size of Proof Tree (uniform (b, d) tree, OR node at root, we win): General pattern for best case:

**Minimax search for win-loss-draw and scores:**

**OR node = MAX node**: .

Booleans: True = win; False = lose; ;

if for at least one i

Maximum of {0, 1}: 1 = win; 0 = lose; ; if for at least one i.

Stop earlier: **1.** We know maximum possible score and one child achieves it. **2.** We have a bound, and only want to know if we can reach at least that bound. We can stop as soon as one child achieves that bound.

**AND node = MIN node**: .

if for all i.

if for all i.

**Minimax value m**: 1st search if we can get v ≥ m; scores; 2nd search if we can get v > m. If 1st search returns a win and 2nd search returns a loss, then m must be the minimax value.

Boolean Searches and Proof Tree: Win with test (v ≥ m): proof tree of our winning strategy achieving at least m. Loss with test (v > m): disproof tree of our win – opponent’s winning strategy – prevent us from getting more than m.

**Alpha-beta search**: : we will avoid this position, since we have a better alternative; : opponent will avoid this position, since they have a better alternative; : opponent can also avoid this position, since they have equally good alternative.

Depth limited Alpha-beta: e.g. Win for X: result changes to win score when win is proven at depth 5 -- Interpretation: black can win in 5 moves and black cannot win in 4 or fewer moves.

**Principal Variation (PV)**: Assume both players follow best play based on the stored values, and the root n is an OR node with minimax value score(n) = m.

Alpha-beta and playing optimally – **OR node:**

.

OR can find at least one child with , and play that m.

Alpha-beta and playing optimally – **AND node:**

.

AND can find at least one child with , and play that m.

If both players play the best moves, then they follow a PV of the search. This is *a move sequence S* with the property that each node in the sequence has a score of m. S is the intersection of P1 and P2 (set of nodes that are in both trees). Even with a depth-limit and heuristic evaluation, a PV exists. All nodes will share the same value as the heuristic evaluation of the last node in the sequence.

**Zobrist Hash codes:** Prepare one random number code[point][color] for each (point, color) combination; Code of state is bitwise logical xor over all points on the board.

**tt entries**: 1. For Boolean negamax, we only needed one bit (True/False). 2. For alpha-beta and iterative deepening, need to store more: best move from this state; search score; flags: exact value or upper or lower bound; search depth; a flag whether it is exact result or heuristic score

**Heuristic alpha-beta search**: **1.** History heuristic (**Jonathan Schaeffer**): keep track of which moves are effective in causing beta cuts in the search. **2.** countermove heuristic (**Uiterwijk**): store good reply to a move.

**Some quiz question:**

If both take the same time, we should prefer a heuristic over an exact solution. (F)

According to Chris Snijders, it takes some quantifiable data from past experiences and some history for “a simple formula” to outperform a human professional.

In Chris Snijders’ experiments, computer models outperform humans in adherence to budget, accuracy of specifications and timeliness delivery.

Computer models can outperform humans because: faster, more accurate, less inconsistent, lack of emotion, no degradation of decision quality over time, more experience.

Some studies have found that face-to-face interviews degrade decision quality. (T)

The computer-assisted decision-making for Long Term Capital Management ran into some initial problems, but it was ultimately very successful. (F)

One advantage of humans is their ability to “see beyond the model”. (T)

**Assignment2 specification:**

**timelimit seconds** (1 <= seconds <= 100): sets the maximum time to use for all following genmove or solve commands, until it is changed by another timelimit command. Default is 1 second.

**solve**: Response is = winner[move]. winner is b, w, draw or unknown. unknown if your solver cannot solve the game within the current time limit. Solving always starts with the current player (toPlay) going first. If the winner is toPlay or if it is a draw, then also write a move that you found that achieves this best possible result. If there are several best moves, then write **any one** of them. If the winner is the opponent or unknown, then do not write any move in your GTP response.

**genmove color**: color is b or w. If the game is not over, the program should use the solver to try to play perfectly. The solver must respect the current time limit as described under timelimit per move above. If the game is not solved within the timelimit, or if toplay is losing, then genmove should just play a random move.