Notes for AI

Peter Heilbo Ratgen

May 4, 2021

Contents

1 Week 5 February - Introduction

1.1 Basics

The course is an introduction to the basics of Artificial Intelligence. We will get an overview of the base of the artificial intelligence methods. We will use python as a programming language. Labs and support will be done in python. Prerequisite to the exam is to complete the homework of the lectures. [1]

You should help each other, but coding should be done individually.

- Thinking humanly
- Acting humanly

  The Turing test is used to test this. The longer a human can be fooled into thing that the human is talking to a human, and not a bot. This could be chatbots acting humanly. You have to test for:

  - Natural language processing
  - Knowledge representation
  - Automaed reasoning
  - Machine learning

  Success depends on deception. Chatbot can use cheap tricks. Mitzuku has recently won for the 5th time. Computers have a had time with multiple choice questions. Eg the "The large ball crashed right through the table because it was made of styrofoam". If you replace "styrofoam" with "steel", then the answer is totally different.

  A better test? A better Turing test, would be one that can be administed and graded by a machine, and more objectivity in the it would not depend of subjectity of humans.

- Thinking rationally

  It is about the idealized or "right" way of thinking. It is hard to describe the world using logical notation. The procedure of applying these local statements and deducing them. We also have a though time dealing with uncertainty, representing the gray areas.

- Acting rationally

  Acting rationally is acting with the goal of achieving the goal, that has been set. Utility is about the goal that has been set, whether it is about, shortest route, least time or fewest changes in a public transit system.

1.2 Successes of AI

- IBM Watson is an AI created by IBM. IBM is one of the companies that has been investing in AI for the longest times.
- Self driving cars is one of the successes of AI. This is an example of a rational acting.
- Natural language processing is also a great improvement, with speech technologies and machine translation.
- Vision, OCR, handwriting recognition and face detection and recognition.
- Mathematics, program solved unsolved conjecture. Also wolfram alpha.
- Games

  Chess(champion beaten in 1997), checkers(solved in 2007), Go (beaten for the first time by a Google AI), Google AI beat top StarCraft players.

- Logistics, scheduling, planning.

  A lot of the advancement are done by the military. In the 1991 Gulf War an AI planned and scheduled for 50,000 vehicles and such.

- Robotics

  Mars rovers, self driving cars, drones, robot soccer, personal robotics.

Exercises will start at 10:20.

Basics Python code is fairly simple and readable. Beginning and ending of blocks is done purely by indentation. We will use the Python Console for trying out examples. Variables can change types throughout the program. A variable can start as a string end as float. We can use + for string concatenation. We can use triples quotes for strings containing both ' and ".

Variables in Python do not have intrinsic types. But assignment does not create copies, but references. References are deleted by the garbage collector, when the reference has passed out of scope. Names cannot start with numbers.

We can have multiple assignments. And swapping vars is easy.

```
>>> x, y = 2, 3
>>> y
3
>>> x, y = y, x
>>> y
2
>>> x
3
```

Sequence types Sequence types are tuples, strings and lists. In a tuple we can have multiple types of variables. Tuples are immutable, such that they cannot be changed after it has been created. Strings are also immutable. Lists are mutable, they can also have mixed types. These sequence types have much syntax in common. If we have to change elements in an immutable tuple or string a new copy has to be created. Lists can be shrunk or expanded as you go. We assign some different variables:

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu
(23, 'abc', 4.56, (2, 3), 'def')
>>> li = ['abc', 34, 4.34, 23]
>>> st = "Hello World"
>>> st
'Hello World'
>>> st = Helllo wordl'
  File "<stdin>", line 1
    st = Helllo wordl'
                     ^
SyntaxError: invalid syntax
>>> st = 'Helllo wordl'
>>> st
'Helllo wordl'
>>> st = """THis is a multiple line
... string that uses triple quotes"""
>>> st
'THis is a multiple line\nstring that uses triple quotes'
```

We can also have negative indexes, such that -1 is the last character:

```
>>> st[-1]
's'
```

```
>>> st[-2]
'e'
```

If want to get the 3 middle elements of the tuple we defined:

```
>>> tu[1:4]
('abc', 4.56, (2, 3))
```

From this we get a copy of the selected part of the tuple. If we do not specify where in the tuple to start, we start from the beginning.

```
>>> tu[:3]
(23, 'abc', 4.56)
```

We can do a copy like this:

```
>>> tu[:]
(23, 'abc', 4.56, (2, 3), 'def')
```

In this example there is a big difference between line 3 and 4.

```
>>> l3 = ['4', '5']
>>> l4 = ['6', '7']
>>> l3 = l4
>>> l3 = l4[:]
```

In line 3 we assign `l3` we assign to the reference to `l4`. In the 4th line we assign `l3` to a copy of `l4`, such that changes in `l4` will not be reflected in `l3`.

We can use the `in` operator to check if we have a substring. We can concat tuples:

```
>>> tu[:] + tu2[:]
(23, 'abc', 4.56, (2, 3), 'def', 12, 'yeet')
>>> tu + tu2
(23, 'abc', 4.56, (2, 3), 'def', 12, 'yeet')
```

Dictionaries Dictionaries can store a mapping between a set of keys and values. We can create a dictionary with:

```
>>> d = {'user' : 'bozo', 'pswd' : 1234}
```

The values can be anything. We can get the value of a key like this:

```
>>> d['user']
'bozo'
```

We can delete a key:value pair like this:

```
>>> del d['pswd']
>>> d
{'user': 'bozo'}
```

2 Week 7 - Intelligent Agents

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. We can say that an agents behaviour is described by its **agent function**. This maps any sequence to an action. This is a mathematical abstraction, the *actual program* is called the **agent program**. [2, p. 3]

We can have many types of agents, these can perceive in many ways: here are some examples.

- Human agent

  Eyes, ears, nose

- Robotic agent

  Cameras, and infrared range finders

- Software agent

  Keystrokes, file contents & network packets.

2.1 Vaccum-cleaner world

A vecuum-cleaner has two locations to take care of: A and B. It has four actions: left, right, such and NoOp. We can have a simple programme for cleaning with this vaccum:

```
if status = Dirty then return Suck
else if Location = A then return Right
else if Location = B then return Left
```

The vacuum-cleaner is a rational agent, it tries to optimize on a performance measure. The choice of performance measure is a critical one. "Doing the right thing" is to always act according to the performance measure. We also call the performace measure the utility function, it is an objetive criteria for the success of an agent's behaviour.

An example of potential performance measures for the vacuum agent.

- amount of dirt cleaned up
- amount of time taken,
- amount of electricity consumed
- amount of noise generated

It would be desirable to have the agent take as little time as possible, but if we only consider this, then it would stand still. But if combined, with eg the amount of dirt cleaned up, then we could have beter operation.

2.2 Autonomy

An autonomous agnet always has the ability to learn and adapt, it can always say "no", it also needs enough built-in knowledge to survive.

Task Environment Specification Problem specification:

- Performace meassure
- Environment
- Actuators
- Sensors

For a autonomous taxi

- Performace meassure

  Safe, fastm, legal

- Environment

  Roads

- Actuators

  Steering wheel

- Sensors

  Cameras, speedometer

For an email spam filter:

- Performace meassure

  Manimizing false positives

- Environment

  User email account

- Actuators

  Mark as spam, let mail through

- Sensors

  Incoming messages

2.3 Environment types

Fully observable vs partially observable In a games, such as FIFA or any game, it is possible to oberserve everything, and perceive all options. If a robot is playing football in the real world, then the perception of the environment is constrained by the input of the sensors eg eyes (you do not know what happens behind your back).

Deterministic vs stochastic A deterministic environment is when the coming events in time is only defined by the current state, and the agent's future actions. Versus when the game is dependent on some type of randomness, whether it be a deck of cards or dice. This is very hard to predict.

Episodic vs sequential Is every step of the agent independent? Or does it depend on a previous sequence? A sequence could be a game, where current decisions depend on a previous sequence of decisions. In a spam filter the sequence of spam mail does not matter. This is a matter of defintion, it is how you see your environment in terms of your decisions. A game could be episodic, if you choose to look upon it as a snapshot.

Static vs dynamic Is the world changing while the agent is thinking. Solving a rubrics cube or chess is static. However self-driving is very dynamic, here things are changing all the time. This a very challenging

Discrete vs continuous Does the environment provid a fixed number of distinct number of states(can they be enumerated)? Where time is a factor, then time is always continuous, if you do not choose to look at time in buckets of seconds or minutes.

Single-agent vs multiagent Is the agent operating by itself?

2.4 Hierarchy of agent types

- Table-driven agents, it's just like a big lookup table.

  The table sizes can become huge. Designing such a table is challenging. We need to think about every single case.

- Simple reflex agent

  This is the vacuum cleaner. For every state given the rules, we find the rule applying to the current state. It selects its action from the current percept only. Implemented through condition-action rules. Example of if-then algorithms. For this to work, then the environment must be fully observable. This wont then work for a self-driving car.

- Agents with memory, internal state to keep track of past states of the world

  We can call this a model-based reflex agent. Internal state: aspects of the environment that cannot be currently observed. This is useful for partially observable environments. In addition to the rule we have a model, a desripton of how the next state depends on current state and action. We also have a state, a description of the current world state the action, is the most recent action.

  We update the state, according to the state, action, percept and the model. When matching a rule, the we take into account the state and the rules.

- Agents with goals

- Utility

3 Week 8 - Uninformed Search

3.1 Formulating search problems

Many problems can be solved through search. Which sequence of actions can get us to the goal state? This can be solved through search. We might also have a performance measure of minimizing time. For an example a GPS is a search problem, which sequence of places gets us to the goal? [3, p. 2]

There can be many different examples of using search as a solution to a problem. Examples are getting somewhere, we can formulate the problem as:

- Start: home
- Goal: destination
- operators: move one block, turn

We can also formulate getting settled in a new apartment as a search problem.

- start: item randomly distributed over the place
- goal: satisfactory arrangement of items
- operators: select item, move item

3.1.1 Searching

A search algorithm will tell us the exact sequence to get us from the start to the goal. Search strategies are important methods for many approaches to problem-solving. Search algorithms is a basis for many optimization and planning methods.

We will consider the problem of designing goal based agent in observable (the agent always knows the current state), deterministic (that each action we take has exactly one outcome), discrete (that we have finite number of actions to choose from), known (such that the agent knows which states are reached by each action) environments[1, p. 66].

State Space Is the initial state, actions and the successor function (or transition model) defines the state space of the problem. The set of all states reachable from initial state by any sequence of actions. This can be represented as a directed graph (nodes are states and the edge between nodes are actions).

Building a goal based agent To build a goal based agent, we must answer the questions[3, p. 15]:

- How to we represent the state of the world?
- What is the goal and how can we recognize it?
- What *relevant* information do we encode to describe states, action and therir effects aand thereby solve the problem.

3.2 Examples of search problems

How we solve a comic-book maze.

- solution fixed sequence of actions
- Search: process of looking for the sequence of of actions that reaches the goal
- Agent can ignore percepts during execution.

The maze problem To solve a maze by searching we examine the components of a search problem. [3, p. 8]

- Initial state
  Entry
- Successor Function
  Is the result of the doing an action in a state
- Goal state
  Goal state is the exit.
- Path cost
  Assume that it is a sum of nonnegative *step costs*.
- Optimal solution: sequence of actions with the lowest path cost.

The optimal solution is the sequence of actions with the *lowest path cost* for reaching the goal.

Vacation in Romania We are on vacation in Romania, currently we are placed in Arad. However our flight leaves Bucharest tomorrow. Now we want to find the best way to get to Bucharest.

- Initial state
  Arad
- Successor function
  These are the possible states we can move to $S(\text{Arad}) = \{\text{Zerind}, \text{Timisoara}, \text{Sibiu}\}$
- Goal state
  Bucharest
- Path cost
  The cost of the path from the start to the goal, is the sum of edge costs.

Example: Romania

- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

- **Initial state**
  - E.g. Arad
- **Successor Function**
  - S(Arad) = {Zerind, Timisoara, Sibiu}
- **Goal state**
  - Bucharest
- **Path Cost**
  - Sum of edge costs

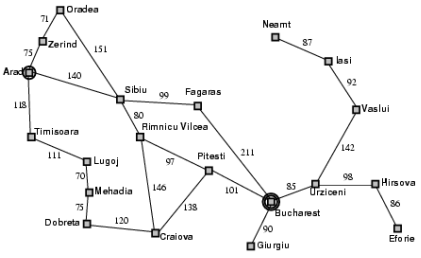

Figure 1: The Romania problem [3, p. 9]

Vaccum-cleaner world We can define a state for the vaccum world:

- Initial state
  Dirty
- Goal state
  All clean

- State space:

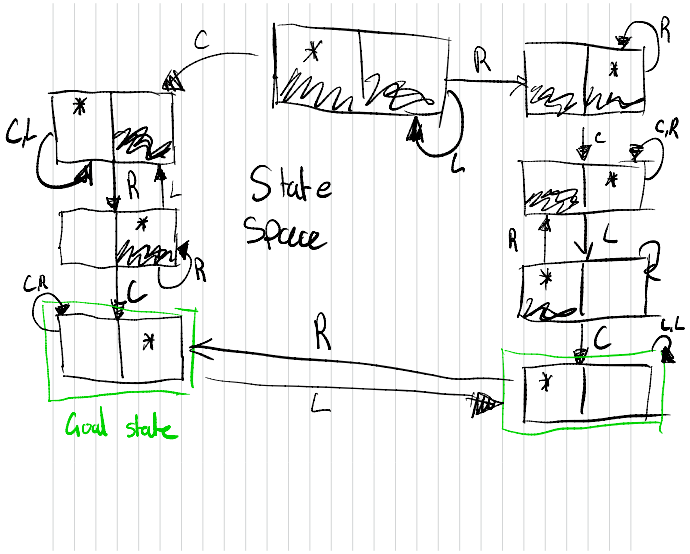

Figure 2: The state space of the vacuum cleaner world

- Path cost
  Could be the sum of the amount of electricity used, or any other type of utility function.

Example: The 8-puzzle The state space of the 8-puzzle has a size of 181440 states. The successor function is the actions of move blank left, right, up, down. The optimal solution is NP-hard[3, p. 17].

Example: The 8-queens problem On chessboard 8 queens are placed. We must place the queens, such that no other queen can attack each other.

- State space: Is the chessboard
- Successor function: depends on how you want to formalize it. It could be moving one piece in a direction, such that you have a different configuration.
- Goal state: Check if any queen is on the same row, column or diagonal.

Example: Remove 5 sticks Remove exactly 5 of the 17 stick such that the result forms 3 squares [3, p. 18].

- States:
- Initial state: The initial state is the state presented with all the sticks laid out, and having the 6 squares.
- Successor function: Picking up any stick from the board.
- Goal test: Check if the remaining sticks form the 3 squares, we need to have a function for that.
- Path cost: It cost one to remove a stick.

3.3 Tree search

We expand at the root node(which is the starting state). As we go about expanding the tree of the search, we maintain a list of unexpanded nodes. This we call the fringe. For each iteration we expand a node we pick from the fringe. We keep expanding until we reach the goal state. Our **goal** is to expand as few states as possible. This is the case as it takes time and can fill up the memory of the computer.

An important point is to that *nodes and states* are not the same. A state is a representation of a physical configuration, while a node is a data structure that is par of the search tree[3, p. 19]

A tree search algorithm works in the following way:

1. Initialize the fringe with the starting state
2. As long a the fringe is not empty:
   (a) Choose a fringe node to expand
   (b) If the node contains the goal state, then return that solution
   (c) If not, expand the node and add the children of this node to the fringe.
3. If we reach the end, terminate having found no solution.

We keep an explored set of the node we already have visited, a node is added to this everytime we expand it. When we add a node to the fringe, then we check if it exists, with a higher path cost. If this is the case, then replace it.

3.4 Search strategies

A search strategy is defined by the order in which we pick which node to expand from the fringe. We have some different parameters by, which to judge a search strategy:

- Completeness
  Does it find a solution if it exists?
- Optimality
  Does it find the cheapest solution?
- Time complexity
  This represents the number of the node generated aka the time it take to run.
- Space complexity
  How many nodes are in memory?

We can measure the time and space complexity in terms of:

- b: maximum branching factor of the search tree
- d: depth of the least cost (cheapest) solution.
- m: maximum length of any path in the state space (could be unbounded).

3.4.1 Breadth-first search

We expand the shallowest node, such that we search in the breath of the tree. This is done by the fringe being made as a first in-first out queue. Such each node that we expand, goes to the end of the fringe list.

Compleness Yes

Optimality Yes - if the cost is 1 for each step.

Time complexity Number of node in a b-ary tree of depth d: the complexity is $O(b^d)$.
Where d is the depth of the optimal solution.

Space complexity $O(b^d)$

3.4.2 Depth-first search

We expand the deepest node first, such that we traverse the depth of the tree, before visiting node higher up. This is done as a last-in-first-out queue.

Compleness Cannot deal with infinte depths, and space with loops. We be complete in finite space, if repeats are excluded from the list of visited nodes.

Optimality No - it returns the first solution that it finds.

Time complexity At depth m: $O(b^m)$.

Space complexity $O(bm)$

3.4.3 Iterative deepening search

This is a mix of depth first, and breadth-first. We first examine the tree in a depth-first with the depth of 1 (we cut off node on level 2 and below). If we do not find out solution we run again with a depth of 2. We again search in a depth first-fashion. It is like having a depth-first approach, but find an optimal solution. This way we cannot fall into infinte loops.

3.4.4 Uniform-cost search

We organise the fringe such that all node are sorted according to the cost. Such that when we expand the root node, then its children B (cost 5) and D (cost 3) are sorted in the order D, B. We then go on to expand D.

Note that [3, p. 33] says that we should keep expanding nodes in the fringe until we are about to expand the goal node, and we should **not** stop when encountering it. Because there might a hidden better solution in the last unexpanded nodes.

4 Week 9 - Informed Search

The idea of informed search is to give the algorithm hints about the desirability of different states. Here we can use an evaluation function to rank node and select the most promising one for expansion. In this way can find the solution (goal) more efficiently, than just using uninformed search[1, p. 92].

We remember from the uniform-cost search, picking the best node (or the most promising node) to expand first. This we call best-first search, where use a evaluation function $f(n)$.

Definition 4.1 *An **evaluation function** is a cost estimate, the evaluation function defines our search strategy (which node we pick first).*

Heuristic function

Definition 4.2 *A **heuristic function** $h(n)$ estimates the cost of reaching a goal from node n. The heuristic function is a type of evaluation function, that is often used in best-first search.*

As an example we can use the heuristic function in the Romaina problem. Presented here with the heuristics:

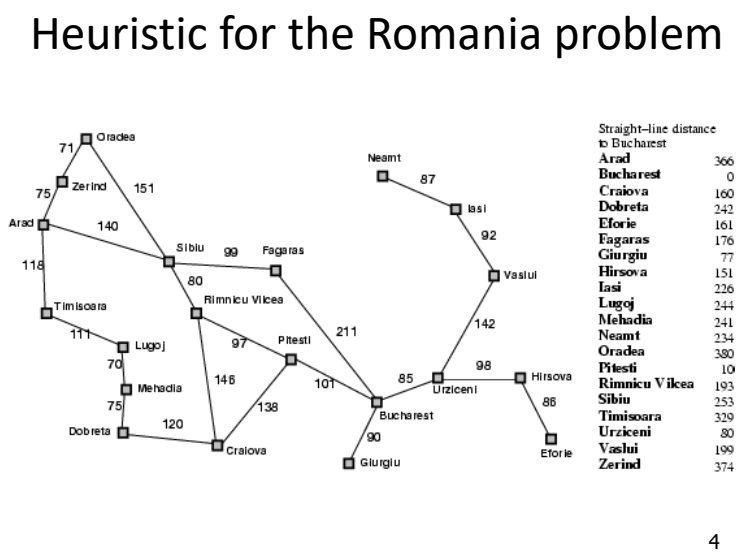

Figure 3: Heuristics for the Romania problem

Figure 3 shows the straight line distance from the town to Bucharest. This is not equal to the shortest distance on the graph. The heuristic is merely to inform our descisions in the heuristic function $h(n)$, when deciding which node to expand.

4.1 Informed Search Algorithms

There multiple types for informed search algorithms. In this section we outline these search algorithms.

4.1.1 Greedy best-first search

Using the greedy best-first search algorithm we expand the node, which has the *lowest value* of the heuristic function $h(n)$.

Properties The greedy-best first search has some properties and pitfalls. The greedy best-first is **not complete** as it can get stuck in loops, where it jumps between two nodes, with a lower value than the two it jumps between. It is **not optimal** as it can make some not optimal descisions. As an example it will make the journey to Bucharest in the order: Arad, Sibiu, Fagaras, Bucharest, while the more optimal solution is: Arad, Sibiu, Rummucu Vilcea, Pitesti, Bucharest. The **time complexity** the worst case is: $O(b^m)$, and best case is $O(bd)$ is the heuristic function is 100% accurate[4, p. 12].

4.1.2 A* search

As the greedy best-first search is flawed, we can change the *evaluation function* $f(n)$. We want to change the evaluation function $f(n)$ to more accurately represent the estimated total cost of the path from the node n to the goal.

Definition 4.3 *We define A* where:*

$$f(n) = g(n) + h(n)$$

- $g(n)$: cost of getting to n, the path cost.
- $h(n)$: estimated cost from n to the goal, this is our heuristic.

The algorithm is identical to uniform-cost, it just uses $g(n) + h(n)$ instead of just $g(n)$[1, p. 93].

Definition 4.4 *An **admissible heuristic** never overestimates the cost to reach the goal (it is optimistic).*

An example of an admissible heuristic is using the straight line distance from n to the goal. This is the case because the straight line distance is always *less* than the true distance to get somewhere. If $h(n)$ is admissible, then **A* is optimal**[4, p. 21]. A* is optimally efficient, no other tree-based algorithm that use the same heuristic can expand fewer nodes and still be guaranteed to find the optimal solution.

4.2 Designing a Heuristic Function

We look at getting a heuristic function for the 8-puzzle.

- $h_1(n)$: number of misplaced tiles.
- $h_2(n)$: total Manhattan distance of the tiles form their goal positions

We now want to determine if the heuristics h_1 and h_2 are admissible. The results of this is for a goal state:

- $h_1(start) = 8$
- $h_2(start) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$

The results of the two heuristics is less than the actual cost, thus the heuristics are optimistic, and this they are admissible.

Definition 4.5 *A **dominant heuristic** the heuristic which dominates the other in A* search. This is the case if $h_2(n) \geq h_1(n)$ for all n.*

Heuristics from relaxed problems

Definition 4.6 *A **relaxed problem** is a problem with fewer restrictions on the actions.*

The optimal cost to nodes in the relaxed problem is an admissible heuristic for the original problem. An example of a relaxed heuristic is the 8-puzzle is relaxed such that a tile can move anywhere on the board, then $h_1(n)$ gives the shortest solution (a tile's shortest path to its right place). We can also relax the rules of the game, such that a tile can move to any adjacent squares, then $h_2(n)$ give the shortest solution.

Looking at the typical search cost for the 8-puzzle (average number of nodes expanded for different solution depths).

If the solution has a depth d of $d = 12$. Then:

- $IDS = 3644035$ nodes
- $A*(h_1) = 227$ nodes
- $A*(h_2) = 73$ nodes

If the solution has a depth d of $d = 24$. Then:

- $IDS = 54,000,000,000$ nodes
- $A*(h_1) = 39,135$ nodes
- $A*(h_2) = 1,641$ nodes

So there is a clear benefit to choosing the right heuristic[4, p. 31].

However if there is no dominant heuristic and we have a collection of admissible heuristics $h_1(n), h_2(n), \ldots, h_m(n)$. Then we can combine them with:

$$h(n) = \max\{h_1(n), h_2(n), \ldots, h_m(n)\}$$

4.3 Weighted A* search

The idea of weighted A* search is to speed up the search at the expense of optimality. So we take an admissible heuristic, and inflate it by $\alpha > 1$, and then person the A* search as usual. By using this approach we expand fewer nodes, but we might not reach the solution in the most optimal way. The cost will be at most α.

5 Week 10 - Local Search

We can formulate problems in terms of optimization. This is that we do not have a start state, and we do not care about the path to a solution. We have an objective function that us about the quality of a possible solution, and we want to find a good solution by minimizing or maximizing the value of the function.

There is three approaches to local search.

5.1 Hill-climbing (greedy) search

Here the crux is: keep a single "current state" and try to locally improve it. You can like it to "climbing mount Everest in thick fog with amnesia".

If we look at a puzzle, it should be 1, 2, 3, 4, 5, 6, 7, 8 going in a clockwise circle our function is

$$f(n) = -(\text{number of tiles out of place})$$

. We want to find the best solution, the one closest to what we think we want.

In each iteration we probe, for which move increases our evaluation function.

The psudeocode the algorithm is:

```
current = starting state
loop:
  next = highest value successor of current
  if (value(next) < value(current))
    return current
  else
    current = next
```

Because it just probes for best solutions in the visible space, it can get stuck in local optima.

An idea for escaping local optima is to iteratively do hill-climbing, each time we initialize with a random condition. If subsequent searches improve, then a stored state is replaced.

5.2 Simulated annealing search

The hill climbing will be never go down the hill of global and local maxima. The idea is to escape local maxima by allowing some "bad" moved but gradually decrease the frequency of these bad moves.

- Probability of takeing downhill move decreases with number of iterations, steepness of downhill move
- Controlled by annealing schedule

Inspired by annealing process, as part of the tempering of glass, metal.

The pseudocode for the simulated annealing search:

```
current = starting state
for i = 1 to infinity
  next = random successor of current
  let delta = value(next) - value(current)
  if (delta > 0)
    current = next
  else
    current = next (with probability exp(delta/T(i)))
```

T is gradually decrease to 0 over some time t. We always accept an improvement (delta greater than 0), but if not we accept with a probability of < 1. We are more likely to pick bad moves early than later.

If the "temperature" T decreases slowly enough, then the simulated annealing search will find a global optimum with a probability approaching one. This is not practical,

the more bad choices we need, the longer the search takes (we make fewer good choices in a row).

A better more modern technique is Markov Chain Monte Carlo algorithm for expolring complicated state spaces.

5.3 Genetic Algorithms

Each living cell contains **chromosomes**, which are strings of DNA. These chromosomes contain **genes**, which are sections of each strand of DNA. The genes determine traits of the organism. A **genotype** is a collection of genes. A collection of traits is called a **phenotype**. Reproduction involdes **recombination** of genes from parents, and then **mutations** (errors) occurs when these genes are copied. **Fitness** of an organism is how much it can reproduce before it dies. We evolve based on "survival of the fittest".[5, p. 21]

A bad idea is to generate random solution and to check if they are correct.

We can take this idea and use it to implement a genetic algorithm (GA).

```
Generate a set of random solutions
while(solution is not good enough)
  test each solution in the set (rank them)
  prune the bad solution from the set
  duplicate the good solutions
    make small changes to these
```

5.3.1 Encoding a solution

We need to encode a solution, such that we can make random changes to these. This is sometimes done as bitstrings (eg. 101011). Where each bit represents some aspect of the problem. An important point is that we need to be able to probe each string and get a score it.

5.3.2 Digging for oil

Along a 1 km stretch of road, we want to choose where to drill for oil. The search space (or state space) is the set of all possible solutions, in this case we have a space of `[0..1000]`. We encode the search space in bitstring of length 10. We then for each generated string check the fitness function, as seen on [5, p. 31].

5.3.3 Search Space

How the space is formed defines the nature of the Genetic Algorithm's performance. As an example a completely random space is bad for a GA, we there is no pattern to it and if we have a lot of local maxima, the algorithm can get stuck in these.

5.3.4 Reproduction in a GA

We look at duplicating the good solutions (ranked according to the fitness function), and making changes to these. Relying on random mutation is not necessarily a good idea. Therefore we introduce crossover in reproduction to get better results.

The idea is: take two high scoring strings and with some probability (the crossover rate) combine them into two new bitstrings. Each of these offspring can also be changed randomly (remember mutations).

In selecting the parents of the bitstrings, we want to pick some of the best strings. One approach is the "Roulette Wheel" approach.

- We start by adding the fitness of the chromosomes.
- Then we generate a random number R in this range.
- Then the first chromosome in the population that is $\geq R$, is picked.

An example of roulette selection is seen here:

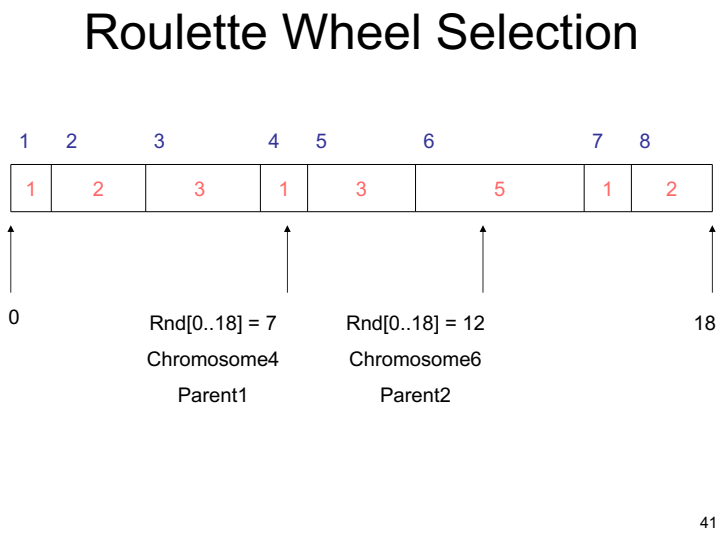

Figure 4: The roulette selection

A way of combine the two parents, when we have picked them is *recombination*.

Definition 5.1 ***Recombination*** *is when we with some high probability* 0.8 *to* 0.95 *choose to replace the rest of the digits, with the content from the other.*

Definition 5.2 *The* ***crossover point*** *is the random single point from the strings are swapped.*

We now also introduce mutations. Mutations occurs for each digit with some small probability between 0.1 and 0.001.

5.3.5 The algorithm

```
Generate a population of random chromosomes
while(solution is not good enough)
  calculate the fitness for each chromosome
  for each chromosome
    use the roulette selection to select pairs
    generate offspring of this with crossover and mutation
  new population as been produced
```

5.3.6 Paramenters of a GA

We of course need to set the population size (the number of chromosomes) the mutation rate m and the crossover rate c. These values have been tuned based on the results. There exists no good general theory to deduce good values. These best solution is the fittest of the population solutions.

6 Week 11 - Adversarial Search

In games there are different types of environments.

	Deterministic	**Stochastic**
Perfect information (fully obeservable)	Chess, checkers, go	Backgammon, monopoly
Imperfect information (partially observable)	Battleships	Scrabble, bridge

Table 1: Types of game environments

We can also call deterministic, fully observable game zero-sum games. A games often has a state space that is easy to represent, as the nature of a game is somewhat restricted. Agents in a game is usually restricted to a small number of action whose outcomes are defined by precise rules.

This is opposition to physical games which has complex descriptions, and the rules can be quite ambiguous. Games are interesting because they are hard problems to solve. The branching factor of chess is about 35. Each player makes about 50 moves. The search tree is therefore 35^{100}. There is about 10^{40} total distinct nodes in the search tree. Games are also interesting because some instate a time limit, such we cannot search for the optimal solution in infinity. We therefore need to make *some* solution, when we cannot calculate the optimal solution.

Definition 6.1 ***Pruning*** *allows us to ignore parts of the search tree, that have no impact on the final choice of the algorithm. Thus reducing complexity.*

6.1 A small game

We can look at two players MIN and MAX. These players each take turns. We can formulate a search problem with the following contents, where s is the state of the game:

- S_0: The initial state of the game
- PLAYER(s): Which player has the ability to move in the game.
- ACTIONS(s): Return the set of legal moves in the current state of the game.
- RESULT(s, a): The transition model, which defines the result of a move a.
- TERMINAL-TEST(s): A terminal test, which is true if the game has ended. States where the game has ended is called terminal states.
- UTILITY(s, p): A utility function defines the numeric value for a game that ends in terminal state s for player p.

In a game ACTIONS(s) and RESULT(s, a) forms the game tree. Each leaf node of the tree has a utility score, as computed by the utility function UTILITY.

6.2 Decisions in games

Since we have a game, with an opponent (in our case MIN), we cannot simply search our way to the result, we are influenced by the actions of our opponent. Therefore MAX must have a contingent strategy.

Definition 6.2 *A* ***contingent strategy*** *is a strategy which is contingent on the moves of an opponent.*

We look at a game tree.

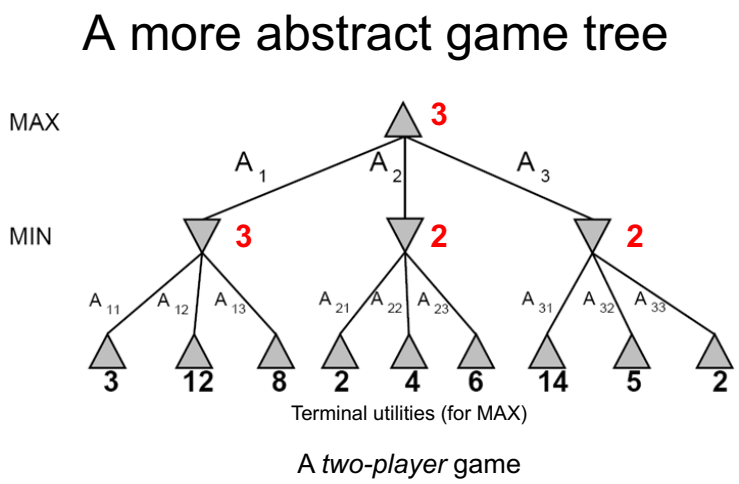

Figure 5: An abstract two player game [6, p. 9]

Here we see that MAX has the possible moves of a_1, a_2, a_3. If MAX chooses a_1, then MIN has the choices of a_{11}, a_{12}, a_{13}. We get the optimality of each node with the MINIMAX(n) function. MAX wants to move to the state with the highest value (maximizing the chance of winning). MIN wants to minimize the value of the MINIMAX(s) function. The logic for the MINIMAX goes:

```
Minimax(state) =
  if (Terminal-Test)
    Utility(state)
  if(Player(s) = MAX) {
    max{Minimax(successors(state))}
  }
  if(Player(s) = MIN {
    min{Minimax(successors(state))}
  }
```

The MINIMAX strategy is optimal against an optimal opponent.

Coin game We have a stack of N coins, each player takes 1, 2 or 3 coins from the stack, the one who takes the last coin loses. The game is defined:

- Initial state: The number of coins in the stack.
- Operators:
  1. Remove one coin
  2. Remove two coins
  3. Remove three coins
- Terminal test: There are no coins left on the stack
- Utility function: F(S)

  F(S) = 1 if MAX wins, 0 if MIN wins

6.3 Alpha-Beta Pruning

The idea is to improve the performance of the minimax algorithm through alpha-beta pruning. A quote is: *"If you have an idea that is surely bad, don't take the time to see how truly awful it is"*.

Through Alpha-Beta pruning we avoid processing the subtrees that will have no effect on the result. We keep two parameters: α, which is the best value for MAX seen so far, and β, which is the best value for MIN. α is used in MIN nodes, and is assigned in MAX nodes. β is used in MAX nodes, and is assigned in MIN nodes.

7 Week 12 - Constraint Satisfaction Problems

8 Week 15 - Probability

9 Week 16 - Bayesian Networks

10 Week 17 - Hidden Markov Models

11 Week 18 - Intro to Machine Learning

12 Lab 1 - Week 7

12.1 TABEL-DRIVEN-AGENT

Run the module The program prints:

The percept

13 Lab 2 - Week 8

1. Successor nodes are inserted at front of the fringe (successor list) as a node is expanded. Is this a breadth (FIFO) or depth-first search (LIFO)?

   This is depth-first search. We put the successor node in front, and examine these first.

2. For goal J, give the fringe (successor list) after expanding each node.

   The fringe list, should be empty, when we have found the goal J. However if we just expand the tree, going depth-first with LIFO then we would have:

   J I H G F C E D B A

3. What is the effect of inserting successor nodes at the end of the fringe as a node is expanded? A depth or breadth-first search?

   The effect of this is a breadth-first search.

4. For goal J, give the fringe (successor list) after expanding each node

   A, B, C, D, E, F , G, H, I, J

14 Lab 3 - Week 9

15 Lab 4 - Week 10

16 Lab 5 - Week 11

17 Lab 6 - Week 12

References

[1] Stuart Russell and Peter Norvig. *Artifical Intelligence. A Modern Approach*. 2016.

[2] Sanja Lazarova-Molnar. *Intelligent Agents*. Presentation. 2021. URL: `../presentations/w7_lec02_agents_sanja_2021.pdf`. (accessed 08.04.21).

[3] Sanja Lazarova-Molnar. *Solving Problem by Searching*. Presentation. 2021. URL: `../presentations/w8_lec03_search_intro_2021.pdf`. (accessed 08.04.21).

[4] Sanja Lazarova-Molnar. *Informed Search*. Presentation. 2021. URL: `../presentations/w9_lec04_informed_search_sanja_2021.pdf`. (accessed 08.04.21).

[5] Sanja Lazarova-Molnar. *Local Search Algorithms*. Presentation. 2021. URL: `../presentations/w10_lec05_local_search_sanja_2021.pdf`. (accessed 09.04.21).

[6] Sanja Lazarova-Molnar. *Games and Adversarial Search*. Presentation. 2021. URL: `../presentations/w11_lec06_adversarial_search_sanja_2021.pdf`. (accessed 09.04.21).