Łukasz Kaiser - Program Search as a Path to Artificial General Intelligence (2007)

Tom Rochette <tom.rochette@coreteks.org>

November 2, 2024 — 36c8eb68

0.1 Context

0.2 Learned in this study

0.3 Things to explore

1 Overview

1.1 1 Intelligence and the Search for Programs

- The core of intelligence is neither the knowledge nor the specific method to use it, but the general way to learn from previous experience
- This is not limited to adopting new knowledge, but also includes learning new ways to use what we know, extending it by reasoning, and even improving learning methods to learn more efficiently
- We claim that the informal notion of a method for solving certain tasks can be expressed in mathematical terms as a Turing machine
- To justify this, we use the Church-Turing thesis, the assumption that everything that is computable, any complex behaviour of a system, can be computed or modelled using only a small set of simple abstract operations
- The thesis of Church and Turing justifies that any informally understood method for solving a problem can be defined as an algorithm, a Turing machine that takes the instance of the problem as input and returns the solution
- We have modeled problem solving as searching for Turing machines with specified properties
- Determining if such a machine exists is of course undecidable and the problem is intractable in general, but we can make some additional assumptions
  - We can assume that we do not only want the machine, but also a proof that it satisfies the formula and that such a machine with a proof exists
  - We will not consider the cases when the problem is not solvable or it cannot be proved that the solution is correct
- Learning amounts to improving the procedure (of the agent), so that after a number of problem instances have been solved it will solve other similar instances more efficiently
- The problem we face with such a theoretical solution is that it would not be usable in practice if implemented in a direct way
  - The time required for it to improve to a level of efficiency that would give any tangible results would be enormous

1.2 2 Theoretical Results

- Let us now consider the Turing machines defined in set theory together with the axioms of set theory as formalized by Zermelo and Fränkel

- Program search problem: Given a formula $\varphi(x_1, ..., x_n)$ in first order logic on the structure defined above (TM and ZFC) with free variables $x_1, ..., x_k$ denoting Turing machines, find a proof of $\varphi(m_1, ..., m_k)$ for some Turing machines $m_1, ..., m_k$.
- Fact 1: There exists an algorithm that computes the solution to the program search problem if any solution exists, so given $\varphi(x_1, ..., x_k)$ it computes $m_1, ..., m_k$ and the proof of $\varphi(m_1, ..., m_k)$, assuming that for some machines such a proof exists.
  - Proof: Since Turing machines, programs, and proofs are enumerable and it can be determined algorithmically whether a sequence of formulas forms a proof of a given claim, we can use the following algorithm to prove this fact:
    1. Set length to 1
    2. Enumerate all k-tuples $m_1, ..., m_k$ of Turing machines shorter than length and all proofs shorter than length and check if there is any proof among these that proves $\varphi(m_1, ..., m_k)$
    3. If the correct machines and proof were found, return them, else increase length by one and return to 2
  - This algorithm is denoted as PSP_0

1.3 2.1 Program Search in the Standard AI Model

- To be able to construct well-acting agents we have to assume something about the environment, or, at least, something about its probabilistic behaviour
- One sensible assumption is that the environment, or at least the probability distribution of events, is driven by some program (TM)
- We want to create an agent that will behave in a worse way than the optimal agent, if one exists, only for some period of time, and that will later act optimally
- Let our agent store the following internal variables:
  - a list of interwoven events and actions called history, intially empty
  - a program model that models the environment, initially any short one
  - a program actor that models the suspected optimal behaviour of the agent, initially any trivial program
  - two numbers max size and max time, initially set to 1
- We consider a model of the environment m_1 to be better than m_2 if we can prove that there is an agent that achieves, using m_1, a better assessment than any agent can achieve using m_2
- When a new event is encountered
  - Append the event to history
  - Search for any program smaller than max size that generates history in less time than max time. Among such environment models, consider only the best ones as defined above, and update model to be one of the shortest of the best programs
  - Search for a proof, shorter than max size, that shows that some program, smaller than max size and halting on every input, can achieve a better assessment in environment model than the program actor. In that case, update actor to be one of the shortest of such programs
  - Increase max time and max size by one
  - Calculate the response of actor to the input event, append the response to history, and output it
- Fact 2: If a Turing machine can describe the behaviour of the environment and there is a provably optimal agent for this environment, then the presented agent gets assessment smaller than the optimal one only for some period of time, and behaves optimally afterwards
  - If the environment is a program, then after some running time it will generate output that distinguishes it from any shorter program
  - Since we assumed that there is a provably optimal agent, this agent and the proof of its optimality have some length
  - When max size exceeds this length, the variable actor will be set to the optimal program. Therefore, the agent will start to behave optimally after detecting the correct environment and the necessary proof

1.4 2.2 Self-improving Program Search

- We do not intend to search for any program in particular, but to learn efficient procedures to search for programs of interest
- Initialize $P = PSP_0$
- Initialize history to an empty sequence
- Divide available resources into two parts and run two processes simultaneously
- Whenever a new instance of a program search problem is received, append it to history
- Main process
  - Receives the problem instance, uses P to solve it and returns the solution
- Improvement process
  1. Append the formula that describes the problem of creating a program search algorithm more efficient than P with respect to μ of the history
  2. Use P to find a more efficient program search algorithm as defined by the above formula (?)
  3. Update P to a new, more efficient version
  4. Repeat, starting from (1) with new P and perhaps an extended history
- If we do not want this algorithm to fall in cycles thinking that some program search algorithm P_1 is better than P_2 and later, when history changes, deciding the other way, we have to assume that the definition of efficiency will be monotonic in some way
- If we are not able to make such assumptions, it could be useful to separate the history of instances received from outside from the self-improvement instances, and use two separate program search algorithms, one for solving the problems and the second to improve program search
- Fact 3: Let a program search algorithm Q (our goal, the efficient algorithm) be given and assume that the efficiency relation is such that there is only a bounded number of algorithms that are provably more efficient than PSP_0 and less efficient than Q, with respect to any possible histories. Then, for any sequence of received instances, the presented algorithm will after some number of steps substitute Q for its internal variable P and therefore become at least as efficient as Q

1.5 2.3 Discussion of Efficiency Definitions

- After gaining experience on a class of instances in the past, we will normally say that an algorithm is efficient if it solves the instances from this class and other similar instances fast
- Two instances are similar if one can be transformed into the other using a few simple transformations, for example by changing some parameters or shifting them in some way
- We can define the level of similarity between two instances as the number of transformations that have to be applied to get from one instance to the other
  - For practical reasons we could assume that if this number is greater than some constant, then the instances are not similar at all
- We can say that one program search algorithm is more efficient than another with respect to a history if it is faster on all instances in the history and on all similar instances
- An alternative definition: The weight of an algorithm A with respect to history H is

$$w(A, H) = \Sigma_{\{i\ similar\ to\ some\ j\ \in\ H\}} time(A, i) \cdot 2^{similarity(i,H)}$$

  - $similarity(i, H)$: the smallest level of similarity between i and any instance from H
  - $time(A, i)$: the time it takes A to solve i

1.6 3 Convenient Model of Computation

- To construct a model, we will concentrate only on two basic operations used in programming, namely the possibility to define and apply functions and the possibility to create compound data types
- Therefore, in our model we will operate on objects that represent some data, e.g. 1, 2, [T, F], and on functions like $+$, $\cdot$, and
- To define functions in this model, we write rules telling how one term should change to another, e.g. T and F → F

- In such rules we can use variables, for example, we can write $x + 0 \to x$
- To avoid terms which do not mean anything, we'll introduce types, such that, for example, 1 will have type int and + will have type int, int → int so we will not be allowed to apply it to the boolean value T
- The model we present is known as term rewriting with polymorphic types
- To define the model, we need the following classes, where arity is always a function that assigns a natural number to each element of the considered set:
  - the infinite enumerate set of type variables, denoted α, β, γ
  - the finite set Γ of type names with arity, denoted T, R, S
  - the infinite enumerable set V of term variables with arity, denoted x, y, z
  - the finite set Θ of constructor names with arity, denoted A, B, C
  - the finite set Σ of function names with arity, denoted f, g, h
- Types: The set of types is defined inductively as the smallest set $\mathcal{G}$ such that
  - each type variable $\alpha \in \mathcal{G}$
  - if $T \in \Gamma$ with arity n and $R_1, ..., R_n \in \Gamma$ then $T(R_1, ..., R_n)$
  - for any number n and types $T_1, ..., T_n \in \mathcal{G}$ and result type $R \in \mathcal{G}$ the functional type $(T_1, ..., T_n \to R) \in \mathcal{G}$
- For example:
  - $\Gamma = \{booleans, lists, pairs\}$
    - where booleans has arity 0, lists has arity 1 and pairs has arity 2
  - The example type E of pairs consisting of a boolean value and a list of any other type can be represented as

$$E = pairs(booleans, lists(\alpha)) \in \mathcal{G}$$

- The set $TVar(T)$ of type variables occurring in a type T is also defined inductively by $TVar(\alpha) = \{\alpha\}$, $TVar(T(R_1, ..., R_n)) = TVar(R_1) \cup ... \cup TVar(R_n)$ and $TVar(T_1, ..., T_n \to R) = TVar(T_1) \cup ... \cup TVar(T_n) \cup TVar(R)$ so $TVar(E) = \{\alpha\}$
- The usual intuition behind types is to view them as labeled trees, therefore we introduce the notion of positions in types
- The set Λ of positions is the set of sequences of positive natural numbers
- By $\lambda \in \Lambda$ we will denote the empty sequence or the top (root) position in the type
- For a given type T and a position p we either say that p does not exist in T, or define the type at position p in T (denoted by $T|_p$) in the following inductive way
  - λ exists in each type and $T|_\lambda = T$
  - $p = (n, q)$ exists in $S = T(R_1, ..., R_m)$ if $m \geq n$ and q exists in R_n and in such case $S|_p = R_n|_q$
  - $p = (n, q)$ exists in $S = T_1, ..., T_m \to R$ if either $m \geq n$ and q exists in T_n and in such case $S|_p = T_n|_q$, or $m + 1 = n$ and q exists in R and then $S|_p = R_q$

2 See also

3 References