Program Synthesis with A*

Bas Jenneboer[1]
Supervisor(s): Sebastijan Dumančić[1] ,
[1]EEMCS, Delft University of Technology, The Netherlands
B.Jenneboer@student.tudelft.nl, S.Dumancic@tudelft.nl

Abstract

Search based synthesis has emerged as a powerful tool in program synthesis, the process of automatically generating implementations for software programs given some form of semantic specification. Search based synthesis involves a search over the space of candidate programs that can be derived from a given grammar. A recently developed new inductive logic programming system called Brute demonstrates how the introduction of example-dependent loss functions can dramatically improve the effectiveness of the search. However, as problem sizes grow its performance drops and at the same time Brute produces programs that are not optimally concise. To overcome this problem, we develop an alternative to Brute that uses A* search and we make it available in an imperative setting (Python). We study to what extent A* search can aid the synthesis of highly accurate and more concise programs in a shorter amount of time using several bench-marking problems, i.e. string manipulation, robot planning and pixel art. We initially find A* to have a higher accuracy. However, when we introduce the use of a new improved heuristic both methods end up with an equally high performance. These results emphasize the importance of the choice of heuristic and that both methods excel in solving distinct problems and can therefore complement each other.

1 Introduction

Program synthesis is the process of automatically generating implementations for software programs given some form of semantic specification. Compared to compilers for which this definition also holds, program synthesizers accept specifications to be given in a much larger variety of forms. Compilers generally require the specification to be embodied by a step-by-step procedure. During translation to machine code, they apply a sequence of local transformations, staying close to the syntactical structure of the procedure in order to preserve semantic equivalence. In contrast, program synthesizers don't require guidance from a procedure to meet the semantic constraints. In fact, the area of program synthesis aims to construct synthesizers that can deal with any kind of constraint, expressed by any logical formula.

This research focuses specifically on inductive program synthesis, the technique of generating programs from descriptions of their desired behaviour. Rather than relying on the completeness of a formal specification, as is the case for its deductive counterpart, this technique is able to induce programs from even a very limited amount of hints. In particular we engage in programming-by-example (PBE), in which these hints are given as input data together with the desired output data. One can imagine PBE offering useful applications on its own. For example, by enabling non-programmers to manipulate large data sets in an automated way, simply by first demonstrating the intended behaviour on a few elements of the data. A less obvious result is that PBE even allows for generating programs from any logical formula, as outlined in the next section.

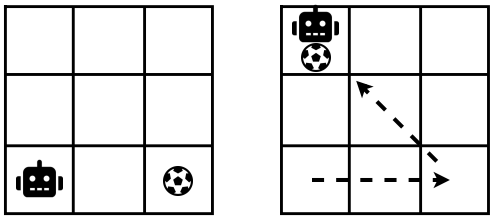

Figure 1: Specification for a robot problem with the input data on the left and the desired output on the right.

A commonly known method for doing PBE is so-called search-based synthesis. This method involves a search over the space of candidate programs that can be derived from a given grammar. A recent study [2] demonstrates how the introduction of example-dependent loss functions can have substantial impact on the effectiveness of search-based synthesis. Traditionally, during the search, candidate program are either rejected after they were found to produce the wrong outputs, or otherwise accepted as the solution program. The authors in [2] developed a new inductive logic programming system called Brute, which extends upon this method by not only rejecting candidate programs, but also taking note of how close their outputs are to being correct. The distance is determined by a loss function that is specific to the type of problem at hand. This information is then used to decide which candidate program to test next.

However, according to [2], as problem sizes are getting larger, Brute more frequently suffers from performance drops. The main explanation for this is the greedy nature of the best-first-search algorithm, which causes an attraction to local minima. The search space has regions where the loss function goes down with respect to the immediate surroundings, but not low enough to reach the global minimum that we aim for if we want a program that satisfies the specifications. Unsuccessfully scanning these regions introduces large delays to the point where the search becomes infeasible. To illustrate this problem, consider the case in which Brute is given the specification in fig. 1.

The specification requires the ball on a grid that is initially on position $(3,1)$ to be moved to position $(1,3)$. Since we can only directly control the movement of the robot, the robot should move to the ball and then carry it to the destination $(1,3)$. As a loss function Brute uses the distance that the robot must travel to its desired destination (without any detours) added to the distance the ball would need to travel (if it were

able to move by itself). In this scenario Brute first produces a program that immediately moves the robot to $(1,3)$, minimizing the loss function as quickly as possible. After this it explores all programs that make this first move exhaustively, before it finally discovers a program that moves the robot to the ball first. In other words, if Brute encounters a local minimum it exhaustively tries to find a solution in that region of the search space, wasting execution time.

Another downside of Brute is that it generates programs that aren't optimally concise. Aside from the programs being less performant, their prediction of the actual intent of the user is suboptimal, since shorter programs are often more generally applicable.

In order to overcome these challenges we have developed an alternative to Brute that uses A* search. The advantage of A* search compared to greedy best-first-search is two-fold:

- A* postpones going down a level of the search tree for longer than best-first-search, since this will add to the path cost. This could make the search less inclined to chase local optima that are contained deep in the search tree and therefore waste time on uselessly scanning a large region.
- A* optimizes for program length. Therefore we are more likely to accurately predict user intent.

This study aims to determine to what extent A* search, guided by domain-specific loss functions, can aid in synthesizing programs that have a higher predictive accuracy compared to a greedy best-first-search strategy in a shorter amount of time. For this, we make the following contributions:

- We make the functionality of Brute available in an imperative setting (Python) as an alternative for the logic programming paradigm that Brute was originally written for.
- We incorporate A* search and run experiments on the same problem domains with the same domain-specific loss functions used by the greedy best-first-search version in [2].
- We test how using better loss functions influences the results.

2 Background

The roots of program synthesis lie in early mathematics on the deduction of constructive proofs from logical specifications, since algorithms can be extracted from these proofs. In practice, the downside of a deductive approach is that users need to come up with logical specifications that are complete and unambiguously reflect their intent such that the desired solution can be deduced from it.

2.1 SyGuS

Arguably, a more useful alternative is inductive synthesis, which recently gained much attention since syntax-guided synthesis (SyGuS) was being popularized by [1] and community efforts to organize and annual competition (SyGuS-Comp). The idea behind SyGuS is that the semantic (logical) specification is supplemented with a syntactic specification, which mostly comes in the form of a grammar. The synthesis challenge is now as follows: Given all the candidate programs derived from the grammar, find one that has the functionality as captured by the semantic specification. Adding syntactic constraints on the programs to be generated has multiple advantages:

- The space of candidate programs can be reduced significantly. In most situations this leads to much more efficient synthesis.
- The possibility to exclude undesired operations from the solutions.
- Minimal languages can be used for obtaining concise implementations, that are easy to interpret by humans.

SyGuS has proved useful in lots of real-word applications. For example, the second advantage can be leveraged for code optimization. Let the semantic specification be the original implementation, that contains expansive operations. By excluding these operations from the grammar a faster alternative can be synthesized. More generally, in theory this technique allows for automatically making existing programs adhere to any new syntactical constraints. This opens the gate to an endless amount of new possibilities.

2.2 Programming-by-example

The third advantage is being leveraged in the area of programming-by-example for very specific problem domains. The power of PBE was demonstrated with the coming of FlashFill in Excel 2013 [3]. Given a column of strings and another column with a handful of manipulated versions thereof, FlashFill first synthesizes a function that implements these manipulations and then applies it to the rest of the column. The synthesized function can be easily interpreted and modified by the user afterwards. The grammar for this problem domain consists of only a small set of string manipulation expressions. Minimal grammars like these are commonly refered to as domain-specific languages (DSL).

2.3 CEGIS

PBE might seem limited to synthesizing from input/output examples at first. However, it was shown by [9] that PBE can be deployed in situations where the specifications equal any logical formula. This strategy for solving general SyGuS problems is called counterexample-guided inductive synthesis (CEGIS). The architecture of the CEGIS solver consists of a Learner and a Verifier. The Learner is a PBE system that receives an initially empty set of input/output examples. The Learner synthesizes a candidate program that satisfies the input/output examples (initally this will be any program that is accepted by the grammar) and passes it to the Verifier. The Verifier then checks the program against the logical formula. If the program does not fully satisfy the specification, a counter-example is produced, i.e. an input/output pair that follows from the logical formula which the program is not able to solve. The counter-example is added to the set of examples and fed back to the Learner. This loop ends when either the Learner can no longer produce a candidate program or the Verifier has acknowledged a program that satisfies the specification. Aside from the Verifier which is out of the scope of this research, the quality of a CEGIS solver is determined heavily by the quality of the PBE solver, which is the focus of this research.

2.4 Search-based Synthesis

Search-based synthesis is a simple, yet popular technique for solving PBE problems. It involves enumerating the vast amount of candidate programs derived from the grammar, until a valid program is found. For this method to be practical, optimization techniques are essential.

- A common practice is reducing the search space by limiting the size of the language. This includes the usage of small subsets of programming languages or even specially crafted minimal languages that are usable for the specific problem domain. DSL's have the additional advantage that the resulting programs are more concise and thus more likely to accurately predict the intended output for future, still unseen input values.
- Another commonly applied optimization is that of pruning parts of the search space based on program equivalence. Suppose we prefer shorter programs over longer ones. If the current candidate program produces the same outputs as and earlier, shorter program for our inputs of interest it is said to be equivalent to the shorter program and therefore ignored.
- Further optimization can be achieved by smart ordering of the search space.

2.5 Brute

The effectiveness of the last mentioned optimization technique was recently demonstrated by [2] with the development of Brute. Brute is capable of learning logic programs much faster than current state-of-the-art inductive logic programming systems by guiding the search with domain-specific loss functions.

However, according to [2], as problem sizes are getting larger the performance of Brute drops. The main explanation for this is the greedy nature of the best-first-search algorithm, which causes an attraction to local minima. The search space has regions where the loss function goes down with respect to the immediate surroundings, but not low enough to reach the global minimum that we aim for if we want a program that satisfies the specifications. Unsuccessfully scanning these regions introduces large delays to the point where the search becomes infeasible.

2.6 A* Search

Another, widely acclaimed informed search strategy is A* search. It uses

$$f(n) = g(n) + h(n) \tag{1}$$

as its loss function, in which $g(n)$ is the cheapest path from *start* to node n found at this time, $h(n)$ is the estimated minimal remaining cost to *goal* from n. $f(n)$ can also be seen as the current estimate of the minimal cost path that goes through n.

Admissibilty

Unlike greedy best-first-search, A* finds cost optimal paths by incorporating preceding path costs in its loss function, that is, as long has the heuristic function $h(n)$ is admissible [7]. A heuristic is said to be admissible if it never overestimates the remaining cost to *goal*. However, even with a non-admissible heuristic A* prefers shorter programs than greedy best-first-search with the same heuristic.

Consistency

A* is optimally efficient in the sense that nodes are never expanded more than once if and only if the heuristic is consistent in addition to being admissible [8]. A heuristic is a said to be consistent if $f(n)$ always increases as n gets closer to *goal*. For an inconsistent heuristic the worst-case time complexity is $\mathcal{O}(2^N)$ due to the risk of re-expansions. However as shown by (cite), if the edge weights in the graph to be searched don't depend on the size of the graph (like in the search over a grammar) the search takes worst-case $\mathcal{O}(N^2)$ time.

Relaxation

A common way of obtaining a heuristic function for combinatorial problems, is by looking at a simplified version of the problem. For example, if a robot on a grid needs to move around obstacles to reach a destination, we can obtain a lower bound on the actual cost by ignoring the obstacles and just taking the *manhattan* distance, which is the shortest path that respects the grid. The situation in which there are no obstacles is a valid simplification, since a solution to the harder problem with obstacles is always a valid solution for simpler problem as well.

As layed out by [5] the enormous improvements on finding optimal solutions to combinatorial problems are the result of better heuristic functions, not better search algorithms. Better heuristics result in search trees with smaller effective depth, not a lower branching factor.

3 Methodology

In our own imperative version of Brute, we also distinguish two stages, invent and search. The invent stage constructs a set of partial programs that serve as building blocks for the search.

3.1 Invent

In analogy to the way in which Brute builds its library, we use *elementary tokens* in order to create more complex *tokens*. By **elementary tokens** (our imperative counterpart of predicates) we mean primitive expressions in the language. We distinguish between two kinds of elementary tokens:

1. **Transition**: When applied to a state it causes a transition to another state e.g. `[MoveUp]` in the robot problem increments the y-coordinate of the robot.
2. **Boolean**: When applied to a state returns `true` or `false`, e.g. `[IsHolding]` in the robot problem returns whether the ball is being held by the robot.

These tokens are composed into two kinds of library tokens by taking their permutations:

1. **Invented**: A sequence of transition tokens, which are sequentially applied to a state.
2. **Control**: Contains boolean and transition tokens for performing some conditional transition, e.g. a while loop or an if-statement.

Pruning

In our case the main purpose of the library is putting further constraints on the DSL for further pruning the search space. For example, we can prevent the creation of

```
If(IsHolding, [MoveLeft, MoveUp, MoveRight])
```

by allowing a maximum of two transition tokens in control bodies. The language is further reduced by disallowing nested control tokens.

Smoothing

Another advantage of the library due to its invented tokens, is smoothing out a rough loss function. Consider the token:

```
[MoveRight]
```

Applying this to the input in fig. 1 results in a decreasing loss function, since the robot moves away from its desired destination. Using only singular tokens the search will have little incentive to explore this direction. If however, the robot would move a little further, i.e.

```
[MoveRight], [MoveRight], [MoveRight]
```

the loss function suddenly increases due to the robot being at the ball. Because of invented tokens, the above program is considered as a single token during the search:

```
[MoveRight, MoveRight, MoveRight]
```

3.2 Search

Programs are constructed by arranging library tokens in a sequence. Roughly the following steps are repeatedly performed during the search:

1. From the current candidate program (initially the empty program) multiple new programs are constructed as a result of appending one of the library tokens.
2. The new candidate programs are being executed on the specified input values.
3. The result is compared to the desired output. If it matches stop iterating, otherwise estimate how close it is to the desired output. Enqueue the program for further extension with a priority based on this estimate.

This process can be seen as traversing an infinite graph, in which every node corresponds to a program and the outgoing edges correspond to tokens that extend the program.

Recall the robot problem from fig. 1. Part of the corresponding search tree is shown in fig. 2. Note that for this specific problem, the programs in both branches produce the same state when applied to the input state. Even though these programs are not equivalent on their own, they are commonly referred to as *equivalent with respect*

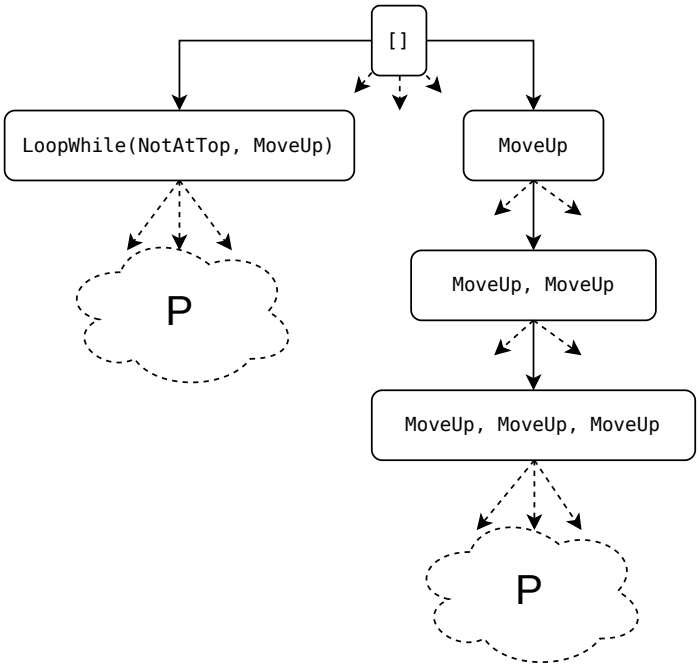

Figure 2: Search tree with program nodes

to the specification. In such a situation it can be justified to discard the higher-cost program, since its future extensions will keep on producing the same outputs as the extensions of the lower-cost program, while all being inferior cost-wise. Recall that applying these programs to the robot example results in a local minimum. Therefore the search wastes time exploring subsequent states, which result from the future programs that are depicted by P. In both branches these subsequent states are the exact same, therefore the search can suffer from the same minimum more than once. One way to avoid this, is to keep track of visited states and costs for reaching them in order to prevent exploring equivalent programs with higher costs, e.g. the right-most branch in the figure.

State nodes

However, our implementation approaches this problem differently (and thereby deviates from the original Brute): The nodes contain states instead of programs (fig. 3). An edge from any node v to w still corresponds to a token that produces the state in w when applied to v. When the desired output state is reached, the solution program is obtained from the edges on the path to the output state. Now equivalent programs with higher costs are implicitly discarded due to standard cycle avoidance. For the situation in fig. 3 this means that only the left-most edge is included in the search tree.

State nodes have another benefit. If the current candidate program $p=[t_1,..,t_n]$ is being extended to $p^*=[t_1,..,t_{n+1}]$, the latter needs to be ran in full on the input states, in order to determine the new distance to the solution. The largest part of the work, which involves applying tokens t_1 up to t_n to the input states, needs to be done twice (once for evaluating p and once for evaluation p^*). By using state nodes, our current node is already the result of executing p. To obtain the result of p^* we only need to apply t_{n+1} to our current node. In cases where program lengths grow linearly with the search tree depth d, this can have a significant effect on the execution time, resulting in time complexity $\mathcal{O}(b)$, instead of $\mathcal{O}(bd)$ for node expansions (with b being the branching factor).

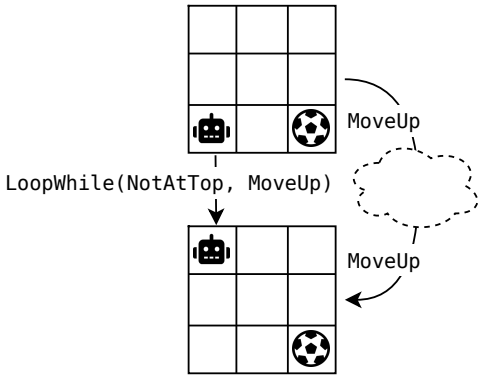

Figure 3: Search tree with state nodes

Compound states

The preceding examples all assumed a single input-output pair. One of the key aspects of search-based PBE is dealing with multiple input-output pairs, producing a single program that satisfies all of them. In part, this is simple to achieve by treating an ordered list of multiple states as a new kind of single state. Formally, if we are given the input states $s_1,..,s_n$, we use the **compound state** $\boldsymbol{s}=(s_1,..,s_n)$ as our input state. Additional rules are:

1. Tokens are applied to compound states element-wise, e.g. $\texttt{MoveUp}(\boldsymbol{s})=(\texttt{MoveUp}(s_1),..,\texttt{MoveUp}(s_n))$.
2. Some state $\boldsymbol{s^*}$ and output state $\boldsymbol{t}$ are equal if they are equal element-wise, that is $s_1=t_1,..,s_n=t_n$.
3. There are several options for computing the distance between $\boldsymbol{s^*}$ and $\boldsymbol{t}$. Brute essentially uses

$$\text{dist}(\boldsymbol{s^*},\boldsymbol{t})=\sum_{i=1}^{n}(\text{dist}(s_i^*,t_i))$$

The last rule gives rise to the interesting question of which aggregation of element-wise distances is most suitable when doing A* search. This could be either taking the sum, the average or the maximum- or minimum value. For the last two options a case can be made:

1. Every individual problem, i.e. computing the distance (s_i^*,t_i) can be regarded as a relaxation of satisfying the parallel problem, i.e. computing the distance $(\boldsymbol{s^*},\boldsymbol{t})$. This is because if we find a single token-sequence that, when applied to compound state s^* correctly returns t, this token-sequence will also correctly return t_i when applied to any s_i^*. Therefore the individual distances are all lower bounds for the compound distance and we can safely take the **maximum** of these lower bounds, in order to obtain the tightest possible lower bound. The more tight our lower bound on the compound distance is, the faster A* will be.
2. The previous argument assumes that we know the individual distances, but we can only obtain distance **estimates** from the original loss functions. As long as these estimates are always lower bounds we are fine, but as we will later show, there are situations in which they overestimate. In order to prevent overestimating the compound distance as much as possible we can take the **minimum** of the lower bounds, to stay on the safe side. By overestimating distances A* is less admissible and therefore more likely to produce program which are non-minimal in length.

Both options are being explored in this study. (In algorithm 1, the first option is shown on line 2).

Loss function

Recall the loss function for A* (eq. 1). The loss (or ***f*-cost**) of node n that the search is currently at, is comprised of two values:

- $h(n)$ is set to the estimated distance to the solution (i.e. the node that contains the ouput states), which is obtained from a heuristic that is specific to the problem domain. For the robot domain for example, the Manhattan distance is such a heuristic. Values of $h(n)$ are commonly referred to as ***h*-cost**.
- $g(n)$ is set to the cost of the path that was taken to the current node. The cost of a single edge is the cost that we assign to the corresponding token. (Composite tokens will be assigned a higher cost than elementary tokens). Values of $g(n)$ are referred to as ***g*-cost**.

As an extension to the traditional A* strategy we accommodate the loss function with weight constant w:

$$f(n) = wg(n) + (w-1)h(n), \tag{2}$$

which allows for convenient switching between the different search strategies by setting w to 0, 0.5 and 1 for selecting greedy best-first-search, A* or Dijkstra respectively. Values in between can also be explored.

This section concludes with explanatory notes on optimization techniques.

Queue updates

Nodes are enqueued with priorities equal to their f-costs. A* can't guarantee cost-optimal solutions with an inadmissible heuristic, because nodes may be visited a second time along a path with lower g-cost and therefore also lower f-cost. Whenever this happens, there is a significant chance that the node is still waiting in the queue with a priority based on the higher g-cost from the first visit. When the node is added a second time together with the lower f-cost, it takes precedence over the old instance. At some point in the future the old instance will cause an unnecessary re-expansion of the node. How often this happens depends on the quality of the heuristic, but it can be avoided altogether with the use of a special priority queue that supports updating priorities. Under the hood this is achieved without adding to the time complexity by keeping node hashes together with references to queue items in a hash table. If a node is re-added, the existing queue item can be dereferenced and marked as *removed*. Removed queue items are immediately discarded when they become dequeued.

Infinite distances

It might so happen that the input-output pairs cannot possibly be satisfied by any program that exists in the language. Since the program space is infinite, this situation is impossible for the search to recognize, but for certain problem domains the heuristic function can be supplied with knowledge on the limitations of the DSL (as demonstrated in the next section for the string domain). When the heuristic function recognizes reaching a solution is impossible it returns an infinite h-cost. The search refrains from enqueuing nodes with infinite h-cost (line 31 in algorithm 1). If the h-cost is infinite for all possible node expansions, the queue becomes empty and the search reports the specification to be unsatisfiable.

Tie-breaking

In practice, the front of the queue often contains multiple nodes with the same f-cost, there is a tie. Enforcing nodes with the lower h-cost to take precedence in case of a tie improves performance of A* in most cases [4]. We've observed experimentally that this is true for our use-case when compared to using no tie-breaking rule at all. This result can be mostly ascribed to the fact that h-cost tie-breaking causes the solution node to be dequeued as soon as it enters the queue. Without tie-breaking, oftentimes numerous nodes with equal f-cost would be expanded first, even though the solution was already in the queue, which lead to more iterations. Our implementation breaks ties by making the queue use h-costs as secondary keys.

Algorithm 1 A*-based inductive program synthesis

Require: $0 \leq w \leq 1$, T : *The set of invented tokens*

```
 1: procedure CompoundDistance(s,t)
 2:     return max_{i=1}^{n} {Distance(s_i,t_i)}
 3: end procedure

 4: procedure LossFunc(g,h)
 5:     return wg+(1−w)h
 6: end procedure

 7: procedure Construct(s,parents,tokens)
 8:     program ← []
 9:     while parents[s] do
10:         prepend tokens[s] to program
11:         s ← parents[s]
12:     end while
13:     return program
14: end procedure

15: procedure Search(s,t)                          ▷ s: inputs, t: outputs
16:     Q ← ∅                                      ▷ queue with priority (f,h)
17:     parents, tokens,g,h,f ← []
18:     g[s] ← 0
19:     h[s] ← CompoundDistance(s,t)
20:     f[s] ← LossFunc(g[s],h[s])
21:     Enqueue(Q,s)
22:     while Q ≠ ∅ do
23:         s ← Dequeue(Q)
24:         if ∀i ∈ [1..n], s_i = t_i then
25:             return Construct(s,parents,tokens)
26:         end if
27:         for all t ∈ T do
28:             s* ← [ApplyToken(s_i) : i ∈ [1..n]]
29:             g* ← g[s] + TokenLength(t)
30:             h* ← CompoundDistance(s*,t)
31:             if h* ≠ ∞ ∧ (s* ∉ g ∨ g* < g[s*]) then
32:                 parents[s*] ← s
33:                 tokens[s*] ← t
34:                 g[s*] ← g*
35:                 h[s*] ← h*
36:                 f[s*] ← LossFunc(g[s*],h[s*])
37:                 Enqueue(Q,s*)                  ▷ Or update priority
38:             end if
39:         end for
40:     end while
41:     return Specification is unsatisfiable
42: end procedure
```

4 Experimental Setup and Results

For doing our measurements, a search-based inductive programming system similar to Brute was implemented in Python. We also made use of the same three problem domains and DSL's, only we used their imperative counterparts. For our measurements we used the same bench-marking tasks that were used in [2]. In all cases we chose the search to timeout after 10 seconds.

Invention

During the invention stage we put further syntactical constraints on the generated programs according to the following grammar:

```
<token> := <ctrl>
         | <trans>
         | <trans1> <trans2>
         | <trans1> <trans2> <trans3>
<ctrl>  := If(<bool>, <trans1>, <trans2>)
         | LoopWhile(<bool>, <trans>)
         | LoopWhile(<bool>, <trans1> <trans2>)
<trans> := varies per problem domain
<bool>  := varies per problem domain
```

We have limited the invented tokens to contain up to three transition tokens. If-conditionals had singular transition-token bodies and while-loop-bodies contained up to three transition-tokens. The following g-costs were assigned:

- cost(transition token) $= 1$
- cost(control token) $= 1 +$ sum of costs for all tokens in the body

Multiple alternatives were tried, but this particular configuration was observed to perform best and was therefore used for all of the measurements that follow.

4.1 String domain

Setup

First we looked at the manipulation of strings by a cursor that moves along the individual characters and modifies them. The string DSL contains the above grammar and is further extended by:

```
<trans> := Move[Left|Right]
         | Make[Lower|Upper]Case
         | Drop
<bool>  := At[Start|End]
         | NotAt[Start|End]
         | Is[Letter|Number|Space]
         | IsNot[Letter|Number|Space]
```

In a addition to using the Levenshtein distance as a heuristic (as was done in [2]) we worked out a better quality heuristic, mainly by using the fact that our DSL only supports removing (not adding) characters.

$$\mathrm{OPT}(i,j) = \begin{cases} j & \text{if } i = 0 \\ \min \begin{cases} \mathrm{OPT}(i-1,j-1) & \text{if } x_i = y_j \\ 1+\mathrm{OPT}(i-1,j-1) & \text{if } x_i \equiv y_j \\ 1+\mathrm{OPT}(i-1,j) & \text{otherwise} \end{cases} & \\ \infty & \text{if } j = 0 \end{cases}$$

This formula optimizes for the following rules:

- For each character in x that is left unmatched, add a cost of 1, since the character can later be dropped with our DSL.
- For each character in y that is left unmatched, add a cost of ∞, since the solution is not reachable with our DSL.
- If x and y are matched and they are equal, don't add any cost.
- If x and y are matched and they are equal except for being lower/upper case, add a cost of 1, since the case can later be toggled with our DSL.

An example of a program that was generated with A*, together with an execution trace for the input *"A-Starrr"* is:

```
1. [MoveRight, Drop]
2. [MoveRight, MakeUppercase]
3. LoopWhile(NotAtEnd, [MoveRight, Drop])
4. LoopWhile(IsNotUppercase, Drop)

A-Starrr  AStarrr  ASTarrr  ASTrr  AST
^         ^        ^        ^    ^
```

We used string transformation tasks from 1 to 300 from [6]. Each task has 10 input-output pairs (**examples**). For each $n \in \{1,2.,n\}$ we sampled n of the examples as **train examples**. These were fed to the synthesizer and the results were verified against the other $10-n$ examples (the **test examples**).

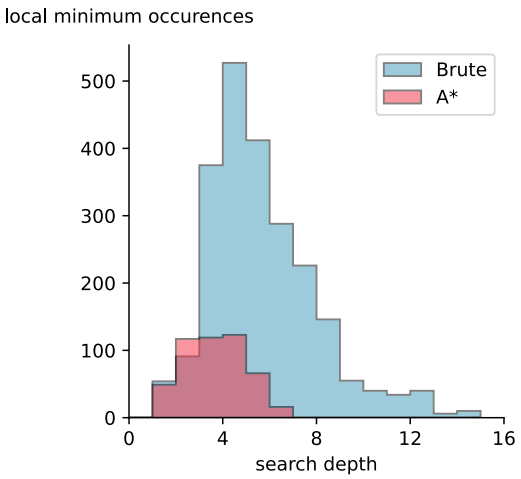

Figure 4: **Comparison of how often local minima are expanded**

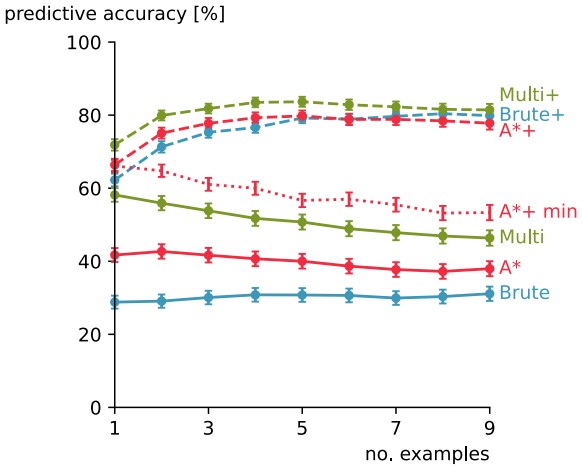

Figure 5: **Predictive accuracies for the string domain**

Results

First of all we could confirm that with A* less local minima were expanded in total over all of the searches (fig. 4). To be precise, we counted every expanded node that was found to have no lower-h-cost children and registered the search depth at which the expansion happened. The histogram shows that Brute expanded significantly more local minima at higher search depths, which is what we expected.

The predictive accuracy (the percentage of test examples that could be solved with the generated program) is shown in fig. 5 (if no program could be generated an accuracy of 0% was assumed). When using the improved heuristic, the search methods *Brute+* and *A*+* have similar accuracies, with Brute even slightly outperforming A* if more than five training examples were used. We use a *"+"* suffix to indicate that the improved heuristic was used. We will later come back to the meaning of *Multi*.

However, with the Levenshtein heuristic on the other hand, A* has a significant edge over Brute. The reason for this contrast is probably that with Levenshtein, the space contains more local minima,

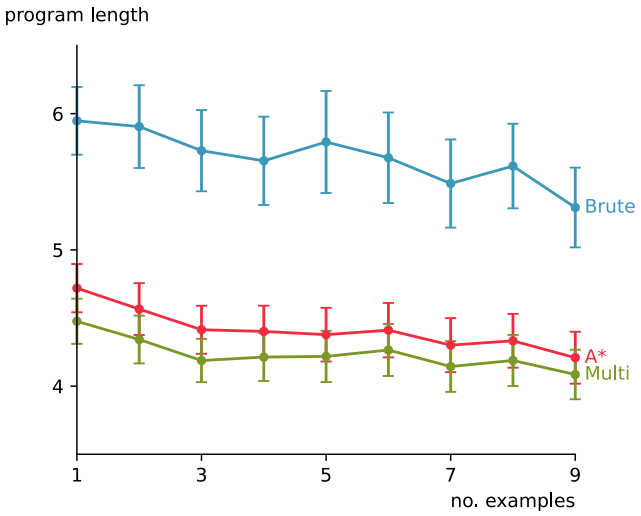

Figure 6: **Program lengths for the string domain**

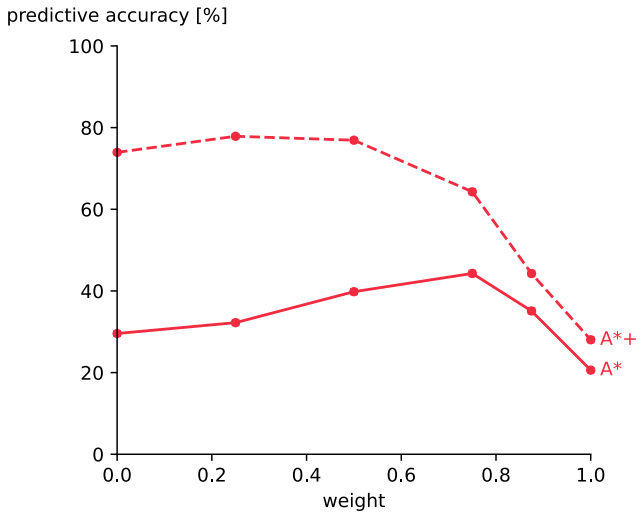

Figure 7: **Predictive accuracy for different weights (string domain)**

which A* is more resilient to. The extra local minima are states with too much characters removed. For example given the train example *"International Business Machines"* → *"IBM"*, Brute initially simply deletes all character, since the empty string is only three characters away from *"IBM"*. A* is often still occupied with exploring lower-g-cost states before visiting these local minima.

The figure also shows the performance when the individual distances of a compound state are aggregated by taking the minimum (see *A*+ min*). It became quite clear that using the maximum, as done by the other plotted searches is the better choice.

We define the length of a program to be equal to the total g-cost of its tokens. In fig. 6 the average lengths of the generated programs are shown. A* generates significantly smaller programs than Brute, which was expected because only A* optimizes for g-cost. However, when using Dijkstra (by setting the w to 0), even shorter programs are generated. This is a consequence of the heuristics being inadmissible, due to the while-loop tokens. Consider the example *"aaa"*→*""*. The h-cost will be 3, which is an overestimation of the actual cost when using `LoopWhile(Drop)` for reaching the solution. We made an attempt at preventing this overestimation (at least for non-nested loops and a maximum of two tokens in the body) by modifying the heuristic to assign a maximum cost to arbitrary length repetitive string modifications, but unfortunately this slowed down the search too much. Still, the programs obtained with A* are often much more natural (i.e. similar to how a human would write code) than with Brute, e.g. for the above *"A-Starrr"*→*AST* example, Brute generates:

```
1. [MoveRight, Drop, MoveLeft]
2. LoopWhile(NotAtEnd, [MoveRight, MakeUppercase])
3. [Drop, MoveLeft, MakeLowercase]
4. [MoveLeft, Drop, MoveRight]
5. [Drop, MakeUppercase, MoveLeft]
6. [MoveRight, Drop, MoveLeft]
```

Figure 5 showed that Brute+ and A*+ solve a near-equal amount of test examples in total. However, instances in which a generated program solves **all** of the test cases of a task were found to occur more often with A*. We suspect this can be attributed to the programs being shorter and therefore likely more general.

Lastly, we ran the experiments with several values for w (fig. 7). The first thing that stands out when Levenshtein is used, is that even more performance can be gained from increasing w even more, compared to $w=0.5$ which is the default for A*. Increasing w increases the priority given to minimizing g-costs (eq. 1). Possibly the performance increase for higher w is caused by avoiding even more local minima. Increasing w too much results in a slower search, with more timeouts leading to a drop in accuracy. Searching with the better heuristic on the other hand, benefits from $w<0.5$. We have found that for some individual w, there can be a significantly large set of test cases that are exclusively solved with that particular w. The *"Multi"* plots in fig. 5 and 6 represent the combined capability of all weights from fig. 7.

4.2 Robot domain

Setup

The second problem was robot planning, in which a robot on a grid needed to move a ball to the desired location. For this problem the transition- and bool rules were replaced by:

```
<trans> := Move[ Left|Right|Up|Down ]
         | Grab
         | Drop
<bool>  := At[Bottom|Top|Left|Right]
         | NotAt[Bottom|Top|Left|Right]
```

An example program that solves the problem in fig. 1 is:

```
1. MoveRight
2. [MoveRight, Grab]
3. LoopWhile(NotAtTop, [MoveLeft, MoveUp])
```

We used all robot planning tasks that were generated by [2]. The tasks only contain a single train example and test examples. The complexity of a task depends on the grid size, which ranges from 2 to 10.

Also to robot problem was provided with an alternative, better heuristic. The original heuristic makes no distinction between the different stages the problem can be in, e.g. whether the robot still needs to find the ball or is already carrying it.

Results

The results are shown in fig. 8, from which we can again conclude that the accuracy only benefits from A* if heuristic is suboptimal.

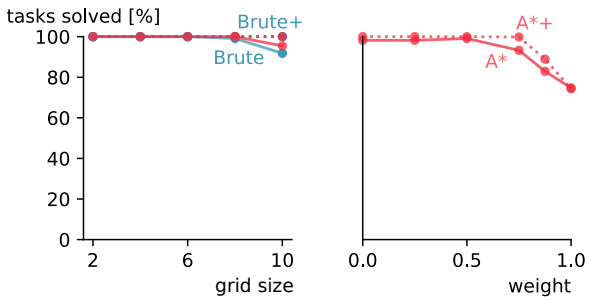

Figure 8: **Predictive accuracy for different problem sizes and weights (robot domain)**

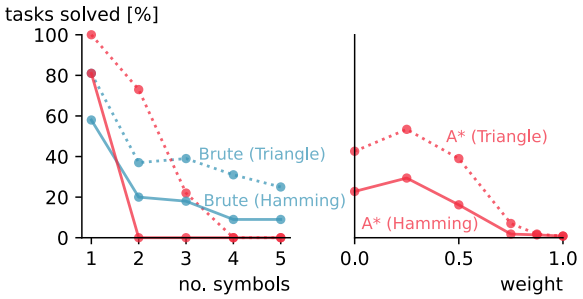

Figure 9: **Predictive accuracy for different problem sizes and weights (pixel domain)**

4.3 Pixel domain

Setup

Our last problem was drawing pixel art. The environment consists of a grid on which a cursor can be moved. Pixels under the cursor can be drawn or erased. We used the grammar:

```
<trans> := Move[Left|Right|Up|Down]
         | Draw
         | Erase
<bool>  := At[Bottom|Top|Left|Right]
         | NotAt[Bottom|Top|Left|Right]
```

The problem complexity is determined by the amount of ASCII characters that are pictured in the desired output canvas. In [2] the Hamming distance is used as a heuristic. Again, we came up with an improvement on this. For the new heuristic we first determine which of the pixels need to be changed. Then we find one of these to-be-changed pixels, say pixel a, that has the largest Manhattan distance from the pixel that the cursor is currently at, pixel c. Their distance already is a lower bound for the distance that needs to be travelled by the cursor. We can make this lower bound even tighter by trying to find a triangle, spanned by the cursor and two to-be-changed pixels, that is as large as possible. Of all the remaining pixels we find the one that has the furthest combined distance to c and a and call it b. It can be shown that a lower bound on the cursor movement is now distance(c,b)+distance(a,b). The results are shown in fig. 9.

5 Conclusions and Future Work

A* can improve the performance of Brute especially when used heuristic is sub-optimal an causes the search space to contain a lot of local minima. In addition, the generated programs are shorter and therefore more likely to correctly predict user intent.

It was also shown that a the weight that is used by A* (0.5), is not always the optimal weight. For future work it can be interesting to adjust the weight dynamically during the search. We had some success with making the weight decrease as we near the time limit, basically imposing a higher risk of visiting local minima in exchange of faster search.

6 Responsible Research

To the best of our believes we think this work is in compliance with the Netherlands Code of Conduct for Academic Practice.

6.1 Acknowledgements

This work has been made possible, not only by the main authors (B. Jenneboer and S. Dumančić), but was conceived in close collaboration with Victor van Wieringen, Farhad Azimzade, Nadia Matulewicz and Stef Rasing. The authors want to thank them for their contributions.

6.2 Reproducibility

We have tried to the best of our capabilities to write down a methodology that can be used by others to reproduce this work. Our implementation of Brute and the A* search method as well as the analysis of the data will be made available at github.com upon publication of this work. Additional data related to this paper may be requested from the authors.

References

[1] Rajeev Alur et al. "Syntax-guided synthesis". In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 1–8. DOI: 10.1109/FMCAD.2013.6679385.

[2] Andrew Cropper and Sebastijan Dumančić. "Learning Large Logic Programs By Going Beyond Entailment". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. International Joint Conferences on Artificial Intelligence Organization, July 2020, pp. 2073–2079. DOI: 10.24963/ijcai.2020/287. URL: https://doi.org/10.24963/ijcai.2020/287.

[3] Sumit Gulwani. "Automating String Processing in Spreadsheets Using Input-Output Examples". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 317–330. ISBN: 9781450304900. DOI: 10.1145/1926385.1926423. URL: https://doi.org/10.1145/1926385.1926423.

[4] Eric A. Hansen and Rong Zhou. "Anytime Heuristic Search". In: *J. Artif. Int. Res.* 28.1 (Mar. 2007), pp. 267–297. ISSN: 1076-9757.

[5] Richard E Korf. "Recent progress in the design and analysis of admissible heuristic functions". In: *International Symposium on Abstraction, Reformulation, and Approximation*. Springer. 2000, pp. 45–55.

[6] Dianhuan Lin et al. "Bias reformulation for one-shot function induction". In: *ECAI*. 2014.

[7] P Russel Norvig and S Artificial Intelligence. *A modern approach*. Prentice Hall Upper Saddle River, NJ, USA: 2002.

[8] J Pearl. "Heuristics: Intelligent search strategies for computer problem solving". In: (Jan. 1984). URL: https://www.osti.gov/biblio/5127296.

[9] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.