

# Automation Algorithms

This appendix evaluates optimization algorithms to provide a background to solver selection in Streamline. The discussion is largely independent of present Streamline: it clarifies options not just for application tailoring as discussed in the thesis, but also for later revisions and wholly different adaptive systems. Understanding of the option space is needed, because optimization models are perpetually amenable to refinement and even relatively minor changes to a model may cause it to fall outside a specific problem class, necessitating the replacement of the solver algorithm. We treat this topic in such detail because, even though decision support is fairly novel in systems design today, we expect it to become increasingly commonplace.

## B.1 Search Algorithms

Optimization is in essence a search problem. The difficulty lies in modeling the problem domain so that a feasible solution can be found within a reasonable timespan. Algorithms can be judged, then, by two selection criteria: quality, in the form of result optimality or bounded distance from optimality, and cost, measured as space and time complexity. Unless in the gigabyte range, we are not concerned about space complexity. Because online operation is envisioned, time complexity is significant, on the other hand. It is discussed in the context of Streamline in Section 6.3.

This section identifies common search algorithms and evaluates them in terms of quality and performance. To render the exposition accessible, it

takes the form of a narrative running from intuitive methods to more formalized, tractable and practical approaches. This overview is strictly a work of synthesis: it does not include any research on our part. It omits experimental or hybrid solutions. More critically, it also excludes string, list, tree and adversarial search algorithms, because these clearly do not match the problem structure of application tailoring.

**Random Search** The most trivial search strategy is repeated random selection. This algorithm, commonly known as a Monte Carlo approach, makes no assumptions on the problem domain. It is extremely robust. The downside to this is that it is also exceptionally inefficient: even recurrent testing of the same option is a possibility. For this algorithm the two sides of the coin are exceptionally well clear. Before we move on to other, “smarter”, alternatives, though, it is important to mention that this trade-off always exists. In other words, "there is no free lunch in optimization" [WM97]. This restriction does not mean that all algorithms are equal when it comes to a given problem, on the contrary. It means that a model builder must understand what kind of structure (pattern) a problem exposes and which solver is most fit to exploit it.

**Greedy Search** A Monte Carlo strategy never removes candidates from the search space. At the opposite end of the spectrum we find a search strategy that always selects the first feasible solution. First fit, or *greedy*, selection trivially prunes the candidate space. For problems with multiple choices, such as path finding, greedy selection prunes the search space by treating each choice as independent, even if choosing one option at choice  $t_n$  constrains the option space at the next step  $t_{n+1}$ . It can be trivially shown that greedy approaches can result in suboptimal solutions. Take, for instance, the network in Figure B.1. An algorithm that selects the absolute cheapest outgoing arc at each vertex (a greedy selection criterion), will never select the arc with a higher cost, even though in the figure this is the only arc that will reach the destination.

The principal advantage of greedy selection is that it is cheap, both in space and time. By definition no global state is kept across choices and no computation is spent on balancing choices. The main drawback, however, is the potential convergence to a suboptimal solution. This is not to say that all greedy algorithms fail to find an optimum consistently. The famous Dijkstra shortest-path algorithm [Dij59] (SPA) is an example greedy algorithm that invariably finds the optimal solution. The algorithm is greedy in that at each step it adds the minimally distant not previously visited vertex to its

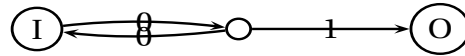


Figure B.1: A network where the shortest arc path traversal strategy fails

set of visited edges. This recurring selection criterion never reduces solution quality, frequently improves it and is known to converge on the best solution at far lower time complexity than random search on average. Its optimality is well known and presented elsewhere [Dij59]. In general, greedy search will return a global optimum if each locally selected optimum is part of this global optimum. If this is the case, the problem is said to have “optimal substructure” [CLRS01].

It is not certain that optimal strategies of the kind seen in SPA can be found – let alone proven – for arbitrary problems. Instead, greedy selection is often based on rules of thumb, or *heuristics*. Such strategies may converge on the optimal solution; for a heuristic to be worth anything it should indeed frequently find the optimum. Nevertheless, the optimality or distance to optimality of a solution generated with heuristics can in general not be established with certainty.

This is not to say that heuristics are not valuable tools, only that used together with greedy search, the combination of early pruning and common sense knowledge can discard good (even optimal) configurations prematurely. Heuristics can be used safely even in the search for optimality. As they direct the search, heuristics can reduce an algorithm’s average time complexity. The A\* algorithm [HNR72] is a generalization of the Dijkstra shortest path algorithm (SPA), where search is directed through an estimate of distance to the goal: a heuristic. Contrary to many heuristic-based searches, however, A\* and SPA have been shown to be complete algorithms: algorithms that are guaranteed to find the optimal solution.

**Logic Programming** A more rigid approach to decision making is to encode domain specific ‘expert’ knowledge in a form that allows synthesis and consultation by a general purpose solver. To automatically derive knowledge from the base set, such *expert systems* must be rooted in a form of formal reasoning. Traditionally, they use deduction to derive relevant statements from a manually supplied set of base facts and relations: the expert knowledge. If

the knowledge base implements a model of an external environment, the approach is also known as *model-based reasoning*. A common approach is to take the Prolog *logic programming* language [Kow74] as starting point for the reasoner, or *inference engine*. In this language, facts and rules are written using human readable labels. As a result, logic programming in Prolog has the appealing quality that facts, rules and even deductions are easily comprehensible. For instance, the following clause states that a path exists between two vertices in a digraph if an arc exists between the two vertices:

```
path(Graph, From, To):- arc(Graph, From, To).
```

As basis for deduction, Prolog implements a Turing complete subset of first order predicate logic. Experts encode their domain knowledge in Horn clauses, disjunction of truth statements with at most a single positive atomic statement (a ‘literal’), e.g.,  $\neg a \vee \neg b \vee c$ , which is logically equivalent to  $(a \wedge b) \rightarrow c$ . Besides rules, the logic can encode simple facts (e.g.,  $d$ ) and goals. If we want to prove  $c$ , for instance, the negated goal  $\neg a \vee \neg b$  must be false for all combinations of  $a$  and  $b$ . In this sentence the statement “for all” hints at the universal quantifier  $\forall$  that is present in (first-order) predicate logic, but absent from proposition logic. For automated reasoning, Horn clauses hold the attractive property that they are compositional, i.e, the resolution of two Horn clauses will always generate another Horn clause. Conversely, each goal clause can be rewritten to give another (set of) clauses that must hold. Reasoning through such resolution can take two forms: *forward chaining* is a data driven method that expands the base set of clauses into all its logical consequences. *Backward chaining* is its goal driven inverse, which takes a goal and tries to establish its truth value by recursively resolving the clauses on which it depends.

From the discussion of chaining operations we can see that logic programming cannot – or not efficiently – model numerical constraints. Incorporating this information in the model is essential to optimize such optimization problems as application tailored I/O. *Constraint logic programming* [JL87] extends inference to include numerical (in)equalities. In CLP, it becomes possible to express, for instance, that a clause only holds subject to a given guard statement, as in this example copied verbatim from the excellent introduction on CLP by Frühwirth [FHKL92], where the righthand constraint limits when the lefthand relationship holds:

```
and(X, Y, Z)  $\Leftrightarrow$  X=0 | Z=0
```

For use in optimization, logic programming has two drawbacks. First, a logic program does not define an objective function to minimize or maximize. Therefore a solver can only perform a first-fit search for a solution in

the feasible region. If multiple solutions can satisfy a goal (if the feasible subspace of the search space is larger than a single point) the first solution that converts the goal into a definite clause will be returned. Introduction of constraints does not change this behavior. For instance, when searching for a path using the goal displayed above, the first returned value is not necessarily the shortest arc. The second well known issue surrounding expert systems is that loosely structured knowledge bases easily results in a combinatorial explosion of the search space. The creator of linear programming (to which we turn our attention shortly), George Dantzig, remarked

“there were a large number of *ad hoc* ground rules issued ... Without such rules, there would have been, in most cases, an astronomical number of feasible solutions to choose from. Incidentally, ‘Expert System’ software which is very much in vogue today makes use of this *ad hoc* ground rule approach.” [Dan02]

For a goal with two free literals  $M$  and  $N$ , the search space grows with  $O(M * N)$ . This is not problematic if each combination of the two literals constitutes an admissible solution – in other words, when the feasible region covers the search space. Few problems are so permissive, however. Ordering facts in the database to be sure that some are tested before others can direct search to spend more cycles on feasible solutions than on infeasible ones on average. Solver behavior is generally not disclosed: logic programming is devised to be non-deterministic, even though the solver clearly is not. Prolog developers must employ such tricks as ordered fact introduction to maintain reasonable search time, but since search behavior is not specified and differs between implementations, this approach of exploiting structure is fragile.

**Exploiting Structure** If the region of infeasible solutions is large, combinatorial state explosion needlessly increases search time complexity – unless the search can be directed to avoid infeasible solutions as much as possible. Note that staying completely within the feasible region is not necessarily the preferred strategy. An optimum may lie at the frontier of the feasible region, in which case search should be focused to the vicinity of this frontier, which may include solutions on the infeasible side.

Application tailoring is a problem where many combinations are infeasible because they have no solution (e.g., matching a filter name to a space that lacks a suitable implementation). The question when modeling this problem then becomes what structure is inherent in the problem and which model with accompanying solver will make most use of this knowledge of the problem domain. For this reason, we now review common structured problems

with known solvers. The goal is to find a structure that matches the problem domain of application tailoring and for which a fast search algorithm is known. In general, the more restrictive a structure is, the more efficient its solver can be, since the restrictions on the feasible region result in fewer tests outside this region. Basic datastructures, such as lists, trees and graphs are all models with well known efficient search algorithms. Mapping a given problem on one of these models is an art, not a science; Section 6.3 will show the implementation of application tailoring as example. That said, some structures can be easily discarded as too constricting. Throughout this dissertation we have depicted requests and systems as digraphs. Mapping these spaces onto lists or trees will not – at least not efficiently – convey all potential applications.

**Dynamic Programming** Divide and conquer is a three step structured search process. One, the solution space  $S$  is split in subspaces  $S_1, S_2, \dots, S_n$  in such a way that the union of the subspaces covers  $S$  – otherwise some parts of  $S$  would not be reachable by the solver. Two, a solution is sought for each subspace. Three, the solutions are combined to select an optimum.

Subspaces need not be disjoint. One example of overlap is where the same structure returns multiple times. Then, extracting this recurrent  $S_r$  once and applying the result repeatedly (*memoization*) speeds up the search. *Dynamic programming* [Bel66, Dre02] identifies repeating patterns in the search space and reapplies the once calculated solution in each instance. The classic problem fit for dynamic programming is calculation of the recursively defined Fibonacci sequence, where each progressive number is defined as the sum of its predecessors:  $x_i = x_{i-1} + x_{i-2}$  with the special cases  $x_0 = x_1 = 0$ . *Branch and Bound* [LD60] is an optimization algorithm that repeatedly splits (or ‘branches’) for another purpose. It prunes the search space by discarding subspaces that are unlikely or even impossible to contain the global optimum. For this purpose the method keeps a global cutoff threshold. This value may change as progressively better solutions are found. In this discussion of greedy search, we want to stress that B&B returns an approximate solution unless we can be certain that the optimum will not be pruned. We will shortly see further application of structure and especially of the branch and bound algorithm.

In application tailoring, subspaces are disjoint nor recurrent: The conflicts between optimization vectors indicate that options are interdependent (Section 6.2.3) and we see little use for repeating (let alone recurrent) subgraphs in practical I/O. Dynamic programming and divide and conquer are therefore not likely approaches for this problem.

**Combinatorial Optimization** Graph matching is a discrete problem, which takes us into the domain of *combinatorial optimization*. Many optimization problems are combinatorial in nature. Some recur so often that they have become archetypical of whole class of similar situations, for instance the traveling salesman. As a result, these classical problems have received considerable research interest, which in many cases lead to the discovery of specialized solvers that outperform more general combinatorial methods such as exhaustive search or branch and bound. One promising approach to efficient application tailoring, therefore, is to reduce the task to a classical problem. We present a selection of intuitively promising candidates – it is infeasible to list and explicitly discard all problems exhaustively here. The selection and similar problems have been frequently presented in more detail elsewhere, for instance by Dantzig and Thapa [DT03].

*Classic Combinatorial Problems* Application tailoring assigns one set of resources to another. The *assignment problem* is a highly abstracted form of this task, that can equally well be applied, for instance, to the scheduling of airline crews to a fixed airline flight schedule. The problem with reduction to an abstract problem is in handling specific constraints. For the flight schedule, it must be ensured that a crew can only fly one flight at a time, must take off from where it landed, and must take a (flight time dependent) rest period between flights. Application tailoring alternates selection of two types of resources (filters and streams) and supports stacking of one. These constraints cannot be mapped perfectly onto the assignment problem.

Assignment is a special case of the *transport problem*, which concerns itself with the movement of goods from a set of producers to a set of consumers. Each producer has a given production capacity, each consumer has a demand and each arc from producer to consumer has a cost factor associated with the transport of a single item of goods. The transport problem is devise a transport schedule that satisfies consumer demand at minimizing transport cost. If we model streams as arcs and filters as vertices, application tailoring begins to resemble transport. Missing, however, are intermediate nodes between data producers and consumers. The *transshipment problem* is a more generic variant of transport that models such intermediate vertices and allows arbitrary arcs between vertices (non-existent arcs are expressed as having infinite cost). Quite a few application tailoring application can be expressed as transshipment problems. The model is not generic, enough, however, to model data manipulation. Many models from classical decision theory, including transshipment, reason about discrete amounts of goods. In computer systems, binary data can be duplicated, split and modified at will.

In application tailoring, we see that streams can be duplicated among all outgoing arcs (fan-out) or split (edge classification). This behavior clashes with the transshipment use of arcs, where arc selection represents choice among shipment options. Filters can also modify blocks and reorder streams. A necessary constraint in transshipment is flow conservation. At each vertex the aggregate input must equal the aggregate output. This constraint will not hold if data modification is modeled. An extension of transshipment that introduces gain/loss factors at the vertices can model such data manipulation. Because factors are static, however, this will result in the modeling of aggregate behavior: a stochastic network. Also, cost is always proportional to the number of shipped goods; static cost (i.e., for arc setup) cannot be modeled. Finally, basic transshipment does not model different types of streams. The *multicommodity* extension models networks that transport multiple types of goods. It replaces the single cost per arc with a vector holding cost for each type of commodity.

Transshipment is a slightly relaxed version of the general *minimum cost flow* problem. In this model, each arc in the transshipment network holds a capacity  $C_{ij}$  (and optionally a similar lower bound). Min-cost is one of three standard network flow optimization problems. The other two are *maximum flow* and *minimum cost maximum flow*. Informally, min-cost aims to maximize efficiency, max-flow to maximize used capacity. Min-cost-max-flow searches for the minimum cost alternative at the maximum flow. Application tailoring typically aims to maximize potential throughput of an application network. It cannot control input, but aims to maximize output data rate given a fixed input rate. The *max flow min cut* theorem states that the maximum flow through a network, where a network is a digraph with capacities set on all arcs, is equal to the total capacity of its minimal cut, where a cut is (informally) a set of arcs that when removed results in an unconnected network. In application tailoring, this statement corresponds to saying that maximum throughput is equivalent to throughput of the bottleneck. Application tailoring is a min-cost-max-flow problem with a task-specific set of constraints. Besides those mentioned before (alternating elements, gains, data duplication and - split), application tailoring has varying data rate (in bytes and blocks) and uncertain costs associated with filters and streams. For this reason, we must yet again look at a more general model.

**Math Programming** All network flow problems – and application tailoring – are exercises in resource planning. A generic planning method originated from research into the automation of military logistic program generation [Dan02]. *Linear programming* (LP) subsumes all presented network

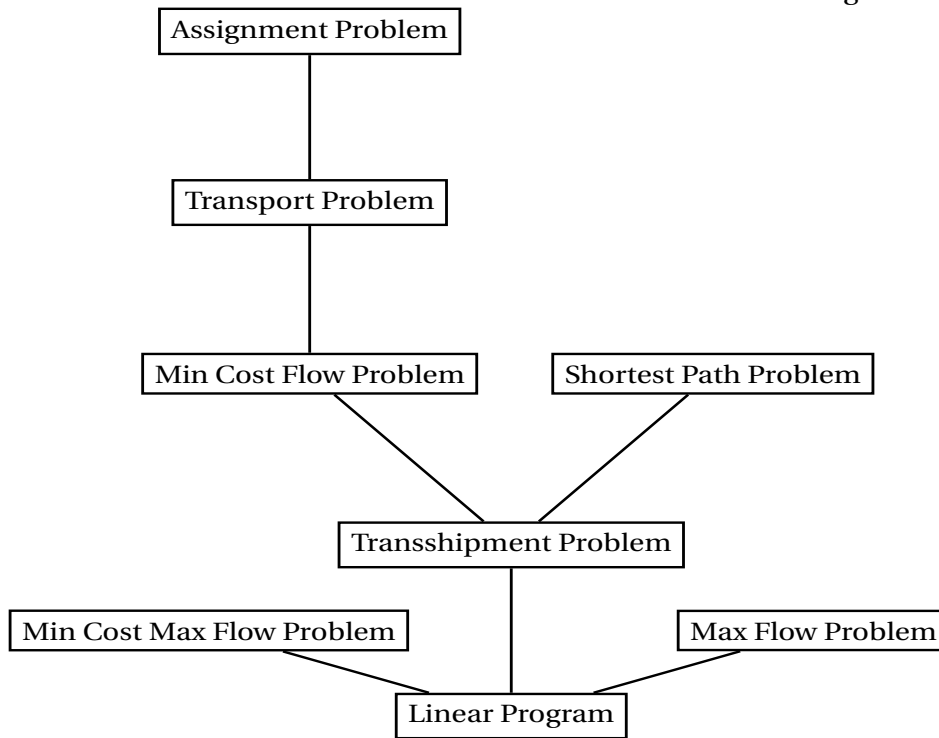


Figure B.2: Some Classic Combinatorial Problems and their Relations

problems and can solve problems with more complex constraints. In LP, a problem is described in terms of a set of linear (in)equalities between variable vector  $x = x_1, \dots, x_n$ . Any combination of values that satisfies all constraints is a feasible solution. Solutions are ranked through an explicit (again linear) objective function over vector  $x$ . The ingenuity of LP is that it enables expedient resolution of the optimal solution by exploiting invariants of the linear solution space.

Let us look at a simple example. Say that a hotel has three rooms, the last two of which can be combined to form a suite. Given four reservation requests, three for a single room and one for the combined suite, we want to calculate the room allocation that maximizes our income. For this simple example, we can immediately see that the only free choice is whether to rent three singles or one single and one double. As the number and types of rooms grows the solution may no longer be so apparent, however. The problem can be described using the following set of constraints, where we introduce the binary decision variables  $x_1, x_2$  and  $x_3$  that decide whether or not to allocate a room and  $x_{23}$  that expresses whether to allocate the suite.

$$x_2 + x_{23} \leq 1 \text{ (we can rent room 2 or the suite)}$$

$$x_3 + x_{23} \leq 1 \text{ (we can rent room 3 or the suite)}$$

In this set of constraints, each variable can be either 0 or 1, denoting occupied or available. In LP, inequalities are usually written more concisely in matrix form  $Ax \leq b$ , giving

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_{23} \end{bmatrix} \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

To rank feasible solution vectors  $x$ , we present an income function  $c$ :

$$[10 \quad 10 \quad 10 \quad 18]$$

The objective is to maximize income by finding an  $x^* \rightarrow \forall i : c^T x^* \geq c^T x_i$ . Because in this objective function the income of the suite is lower than that of the two rooms rented out independently and reservation requests for all three rooms, allocation of the three single requests will maximize income.

Linear programs can be solved much more quickly than loosely structured problems, enabling practical – in our case, online – searching of much larger problem spaces. The central structure in linear programs that ensures efficient convergence to a global optimum is *convexity*. The feasible region of a linear search space is either empty, unbounded or a convex polytope (in two dimensions: a polygon), because each (in)equality splits the search space linearly in a feasible and unfeasible subspace. The feasible region is the intersection of all feasible subspaces. Because the cost (or income) is also a linear combination of variables, each objective value forms a hyperplane. The goal of LP is to find the highest income or lowest cost hyperplane that intersects the feasible region. Feasible solutions are those that have their hyperplane intersect or touch the feasible region. Therefore, a (not necessarily *the*) optimum must always lie on the outer edge of the convex hull. The first LP solver, the *simplex* method, on average converges to the globally optimal solution much faster than a random solver, by walking the exterior of the hull from an arbitrary solution to the nearest local optimum. Since the hull is convex, any local optimum is necessarily also a global optimum.

The invariants that guarantee fast convergence with simplex also restrict applicability of LP. To generate a closed convex hull, the search space must be a continuum. As the example above shows, many practical problems are discrete. Some, but not all, non linear programs can also be solved efficiently. The success of LP in solving complex problems spawned interest in solvers

for other domains, most importantly non linear, discrete (integer) and uncertain (stochastic) problem spaces. These methods are collectively known under the umbrella term *math programming*. Some classes of problems are also inherently convex and have known fast solvers. Others must make use of slower exhaustive search, approximate search or branch and bound. Perhaps counter intuitively, integer problems (IP) can take orders of magnitude longer to solve due to loss of convexity in the search space. One solution to speed up discrete search is to “relax” the problem into a continuous counterpart. Then, simplex will give the relaxed optimum, from which one can deduce an upper bound on the integer solution. Moreover, if the linear optimum happens to lie on an integer coordinate, the solution is certain to be optimal also for the original IP problem.

For some well behaved discrete problems, convexity of this form is guaranteed, such as Network flow problems. These can be expressed as integer problems in the domain  $-1, 0, 1$ . The constraints model the arcs in the network as an incidence matrix, where the beginning of an arc consumes a good ( $-1$ ) and the end produces one ( $1$ ). For these problems a specialized solver, *network simplex*, outperforms even the simplex method. Practical math programming packages can automatically identify subproblems that expose network flow or another structure amenable to fast search and apply a faster solver. In application tailoring, key to curbing time complexity is to expose network structure wherever possible.

One last point of concern when applying LP especially in a dynamic setting is that each change in the model requires a complete recalculation. Because effects on the feasible region are unknown, partial results cannot be safely reused. Cost of recurrent simplex execution therefore scales linearly. For some problem domains, more scalable (in this regard) solvers are known. Dijkstra, for instance, localizes changes and required recalculation to a part of the graph – enabling reuse of other known distances.

**Approximate Search** Sometimes, even fast solvers cannot produce an optimal answer in a feasible timespan. This may be because too little structure can be exploited or because too many options exist. In either case, another – faster – solver must be sought that produces a solutions as close as possible to the optimum in a given timespan: an *approximate search* algorithm. Ideally, an approximate solution can be compared to a known upper bound, as is the case between the linear relaxation of an IP and its integer approximation. Greedy algorithms converge quickly, but in general give no claim on result quality. Besides domain-specific heuristics, they can be said to always follow the *metaheuristic* that the first fit is a good enough fit. In gen-

eral, approximate algorithms employ such model-independent metaheuristics to guide search in an unknown search space. Approximate solvers can be split in three disjoint approaches: Monte Carlo search, local search and population based search. We have already discussed the first. The other two approaches are far more promising if the search space can be expected to exhibit some structure (but not otherwise, the free lunch theorem teaches us [WM97]).

Local search moves from a given position in the search space towards the expected location of the optimum and uses information gained during the walk to alter its direction and speed. The simplest strategy, *random walk* does not apply any strategy to select its next move. A directed strategy is to move (for minimization) in the direction with the steepest decline. Such *gradient descent* (or, hill climbing, in the context of maximization) will converge on a near extreme. In a convex search space, gradient descent will find the optimal. The more erratic the search space, however, the higher the chance that it gets stuck in a local extreme that is not at all optimal. All other approximate search methods make use of gradient descent in one way or another, but they reduce the risk of getting stuck by introducing some risk that they move away from a local optimum. How to trade off the two cannot be said decisively: this is known as the exploration versus exploitation dilemma in search. If the nearest local optimum turns out to be the global optimum, ‘smart’ strategies will generally take longer to converge than simple hill climbing (in line with the free lunch theorem).

The chance of becoming stuck prematurely can be reduced by matching the stepsize to the amount of structural knowledge. Initially, when little is known about the problem, solution quality will easily improve, encouraging the taking of huge steps to explore a wide landscape. As search progresses, however, it is likely that it takes more effort to find improvements. At this point exploitation of known fruitful regions takes over. In local search, where movement follows a single trajectory, the strategy of slowly reducing stepsize is known as *simulated annealing* (SA) – after the physical process of annealing in metallurgy where a material is first heated and then cooled in controlled fashion for structural improvement. In SA, maximum stepsize is a function of a variable  $T$  (for temperature) that starts large, but gradually reduces as the algorithm progresses ( $T$  must always remain positive). At each step, the next stepsize is chosen as a result of  $T$  and the relative improvement seen in the last step.

Local search has little sense of history; another class of approximate algorithms replaces step-by-step operation with parallel exploration of the search space. *Population based search* has a smaller chance of becoming stuck in a local extreme, because multiple agents move more or less independently

of one another. Clearly, population based methods trade off specificity (exploitation) for robustness (exploration). At its simplest, population based search parallelizes local search. Real algorithms are more interesting, however, in that they exchange information between individuals. The manner in which they share results is the main differentiating feature between algorithms. Research continues to produce new approaches, many of which are modeled after physical processes, such as particle swarm optimization [KE95] (derived from social flocking) and ant colony optimization [DMC96]. We here only introduce the most common kind and one influential extension. *Evolutionary algorithms* are population based methods that robustly adapt a population of solution, or ‘species’, in a manner reminiscent of how physical species adapt to their surroundings according to the theory of evolution. Implementation details vary widely. The most common and classical form of EA, *genetic algorithms* (GA) separate search into ‘generations’. At each generation they produce a set of candidate solutions with the help of three operations: selection, combination and mutation. Initially, a random set of individual solutions is created. Each is ranked according to the solution space’s objective function. Then, a selection phase chooses a subset for mating. The simplest strategy only selects the fittest individuals for survival. Experimentation showed, however, that this strategy can lead to premature convergence to a local minimum. More varied and robust selection operations are therefore also commonly applied. Then, a new set of individuals is generated by combining the selected individuals from the last generation. In a pure GA, combination is modeled after chromosomal crossover in DNA (genetic recombination). Finally, for added robustness, random mutation is applied to the DNA strands. The approach requires that a solution vector can be expressed as a reasonably long character string (although not at all as long as physical DNA). Many other EAs exist besides pure GAs, with varying preference for exploitation and exploration. Memetic algorithms (MA) combine GAs at the global level with local search performed by each individual. This approach takes a cue from the Lamarckism – the disproved evolutionary notion that behavior of an individual (phenotype) can influence its DNA (genotype) and thereby its offspring.

**Machine Learning** All approximate search algorithms discussed so far demand that problems have a clear objective function that ranks each candidate solution unambiguously. This is not always the case. There exists a class of problems for which we can reasonably expect some pattern to exist in the problem space, but where we have no way of quantifying this pattern *a priori*. *Machine learning* algorithms try to extract structure during search [Sol75].

The approach is very robust, because it places very few constraints on problem shape. It is particularly suited to problems with dirty and imprecise input data. They are commonly applied to classification tasks, for instance approximate object recognition in (blurry) photographs.

We discern two types of learning: *supervised* and *unsupervised*. In the first, an algorithm learns problem structure by trial. A common approach is *reinforcement learning*, where a supervisor tutors using a training set of data and answers. The algorithm classifies each item in the set and the supervisor skews future behavior by rewarding correct or near correct answers and punishing mistakes – the simplistic carrot and stick approach to education. In application tailoring, reinforcement learning with a domain expert can prepare a self-adaptive system for autonomous decision making. *Unsupervised learning* is not concerned with ranking solutions; it is commonly applied to clustering problems and less applicable to the kind of optimization problems we are interested in here.

Machine learning approaches are well suited to problems where it is difficult to give a clear system model (including an objective function), but offer little other benefit. The most well known machine learning model is the artificial perceptron, or artificial *neural network* [HKP91], with its many variants. Modeling of internal system state is not necessary with machine learning, but input to the model must still be chosen and the output interpreted.

### **B.1.1 Classification**

Figure B.3 summarizes the presented search strategies and classifies them by the problem structure they target. If no structure exists at all, no method will outperform simple Monte Carlo as long as it does not visit any nodes more than once. If structure can be assumed to exist, we differentiate three classes of problems. The simplest class consists of problems of which initially nothing is known, least of all an objective function to rank solutions. Here, supervised learning can help encode subjective ‘expert’ preferences. In the second class, a quantitative ranking function exists, but nothing is known about the relationship between its input and output. Problem structure can be highly irregular, so that exhaustive search is the only approach that can find an optimum with certainty. For large search spaces, exhaustive search is infeasible. Here it is replaced with greedy or approximate search as that converges to higher ranking solutions for a given timespan. The last category consists of problems that expose well known structure, facilitating directed search towards a global optimum.

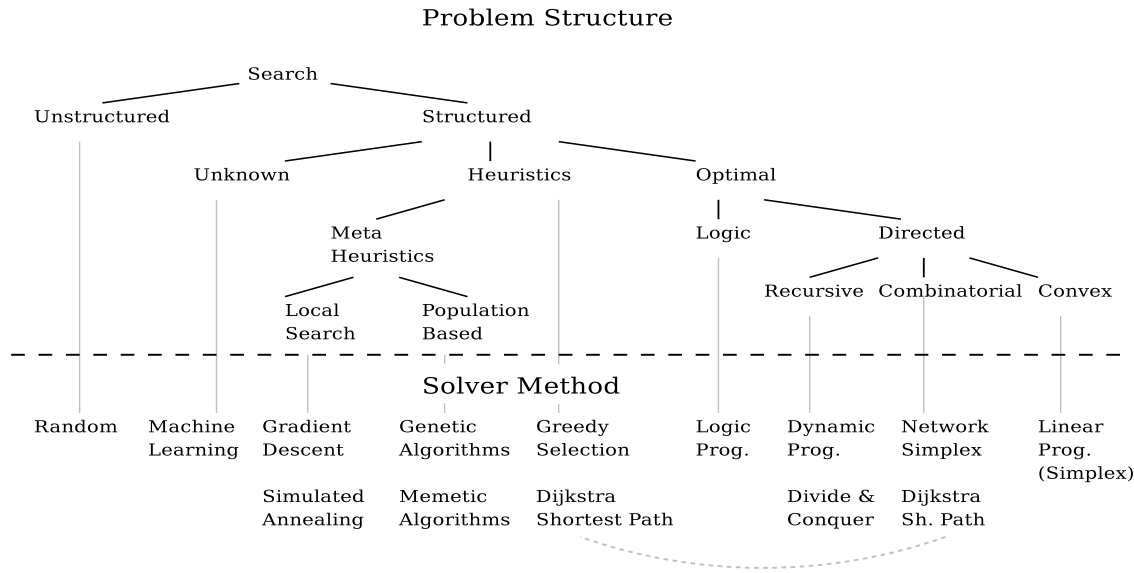


Figure B.3: Classification of Solvers by Problem Domain

## B.2 Practical Considerations In Automation

Two complications render perfect optimization infeasible and therefore application tailoring inexact. First, some properties, such as security, are not easily expressed as a simple scalar value. Second, applications may demand contradicting combinations of properties, such as throughput and reliability. Because models can be refined ad infinitum, it is important to keep in mind that our goal is not to achieve perfect correspondence with reality, but to generate very good (preferable, but not necessarily optimal) configurations.

### B.2.1 Uncertainty

Numerical optimization's Achilles' heel is the assumption that all input can be properly quantified. In reality, model input can often only be estimated – if at all: omissions in data are also probable. To be practical, a control system must be able to generate adequate configurations with such *imprecise* and *incomplete* input. Strict logic parsing cannot handle information gaps or express uncertainty. This section reviews alternatives that are more robust to operation under uncertainty. Called for in this regard is that an algorithm is stable across a wide band of noisy input: that small variations in input do not

result in large perturbations in output. Offline simulation with input variation can generate experimental data on a model's stability to data volatility.

In a model that accepts imprecise input, the difference between quantitative data and heuristics is no longer black and white. Like robustness, admissibility of heuristics can be verified offline. In this manner, a rule of thumb is compared against numerical optimization on a wide array of simulated systems for which perfect input can be assumed. Or, inversely, invariants on large sets of systems and applications can be automatically extracted from simulation, giving heuristics that can reasonably be called admissible.

Uncertainty in model building is a well known problem and many workarounds have been devised. The simplest approach to handle information gaps is to fall back on default values. If values such as cost or capacity are normalized, defining defaults is especially straightforward. Application tailoring picks default values to describe object (filter, stream, space, etcetera) features if none are specified explicitly. Information gaps can be also be seen as an extreme form of imprecision: if a value by definition falls in a normalized domain (say, the unit interval  $[0, 1]$ ), then an unknown value can be said to be at least zero and at most one. Imprecision therefore is the more general problem. We briefly review five approaches of modeling it.

Probability theory [Kol56] takes imprecision as a belief statement. The unit interval is split into regions, each with a probability that the true value of an estimated parameter lies in this region. Bayes' theorem enables inference from such a probabilistic data set. The method requires defining prior probabilities on all statements and is not easily combined with aforementioned solvers. Most importantly, the concept of probability does not really correspond to our notion of imprecision. Dempster-Shafer theory takes a different approach. It models a statement and its certainty by two variables that bound the chance that the statement is true. Belief  $B(A)$  implements the lower bound: if it is one the statement must be true; plausibility  $Pl(A)$  implements the upper bound: if it is zero the statement must be false. Clearly,  $B(A) \leq P(A) \leq Pl(A)$ . Like Bayesian inference, Dempster-Shafer theory supplies a rule to calculate conditional probabilities. It also shares characteristics with imprecise probability modeling. Here, the uncertainty is not encoded by a single probability value. Possibility theory [Zad99] models uncertainty with the upper and lower bounds *necessity* and *possibility*, giving a probability function with the unit interval as domain and range. Possibility theory is not compositional: the probability function of unions and intersections of variables is undefined. *Fuzzy logic* extends possibility theory with arbitrary probability functions and a composition rule for Boolean operations.

*Stochastic programming* [Dan04] joins stochastic data with math programming. Common approaches of stochastic program modeling involves sam-

pling the probability function of a parameter and calculating the optimum for each sample. In this approach, the state space grows linear with the number of samples. If the probability function can be assumed to be monotonic between sampled points, sampling can be curtailed as long as both inputs generate a very similar (for some quantification of very) configuration. Additional sampling points can be generated on demand for problems that show wide perturbations in output. Although the method allows for use of complex fuzzy functions, in application tailoring we limit the uncertainty model to a pair of upper and lower bound estimators for each object. Sampling is only performed to estimate configuration robustness.

### B.2.2 Conflict

Another common issue in optimization is the need in practice to balance conflicting goals. In application tailoring, for instance, we frequently observe conflict between maximizing computational efficiency of a filter and minimizing transport cost of the incident streams. In this case, the variables governing choice are interdependent, i.e., selection of one restricts freedom for the others. Buffer stack selection is an example of largely independent choice, as most buffer implementations can be combined at will (although we have identified a handful of exclusive choices).

Interdependent variables are one type of conflict, another occurs when multiple objectives are defined. Application tailoring as described in the introduction strives for high throughput, but adaptive systems in general have more demands. Streamline enables application-tailored objective functions. One example use is to value high throughput, but demand fault tolerance. More technical goals are isolation from other tasks (real-time constraints), space complexity minimization (embedded operation) and user-level processing (robustness). The interface section lists more options and explains how users can convey demands. This section is concerned with how demands can be reconciled when they conflict. When interdependent variables can be expressed using the same metric, their conflict is resolved by the solver. Conflict resolution becomes problematic when goals are not quantitatively comparable. In this case, a decision maker must be appointed to rank goals. This task is generally ad hoc and subjective. Because in application tailoring we expressly want to avoid involving end users, we must find an automated solution that will adequately trade off arbitrary unforeseen combinations of goals. We briefly review established strategies for structured (*cf.* ad hoc) and preferably objective (*cf.* subjective) conflict resolution.

A first structured approach, multi criteria decision analysis (MCDA), makes implicit preferences explicit by ranking all options along a number of axes.

Choices that are clearly suboptimal on all dimensions are then safely purged. To decide among the remaining candidates, a subjective last selection step is still required, however. These methods are commonly applied in unquantifiable decision making, e.g., public policy making. In numerical optimization, the same problem is known as *multi-objective optimization*. Within this domain, the relaxation of math programming with multiple objectives is referred to as *goal programming* [CC61]. At its simplest, a goal program is a linear program with multiple objectives  $c_i$  that are in principal incomparable.

To balance goals, they must be brought into correspondence. While in practice many methods are ad hoc, this is not to say that more structured approaches do not exist. Subjective balancing takes one of two main forms: *lexicographic ordering* or *weighted combination*. Multi objective solutions can be written as a vector  $x_i = (x_{i1}, x_{i2}, \dots, x_{in})$  of all objective functions' outcomes. In lexicographic ordering, solution raking is strictly ordered by ranking of the ordered scalar results. Solution  $x_i < x_j \iff x_{i1} < x_{j1}$ . If  $x_{i1} = x_{j1}$ , then the next scalar pair  $x_{i2}$  and  $x_{j2}$  is compared, etcetera. Intuitively, it is easy to think of lexicographic order as the alphabetic order used in dictionaries and phone books. Besides pure lexicographic order, where objectives are totally ordered, forms can be constructed where some objectives are equally strong. For instance, throughput may be considered of principal importance, but isolation and fault-tolerance equally important secondary concerns. In this case, objectives are no longer totally ordered. Now the solution space is only partially ordered and can contain mutually incomparable solutions.

Weighted combination is the alternative to lexicographic ordering that defines a single function  $f(x_{i1}, x_{i2}, \dots, x_{in}) : \mathbb{R}_n \rightarrow \mathbb{R}$  that combines all objective values into a single equation. The simplest form is as a linear weighted sum

$$f(\vec{x}_i) = \sum_{j=1}^n \alpha_j x_{ij} - \beta_j$$

where the weights  $\alpha_j$  scale ranges and the offsets  $\beta_j$  transpose the origins, to make dimensions correspondent. Like the ordering of scalars, definition of weights and offsets is principally subjective. One structured method is not to give preference to any objective. In that case, each scalar is normalized, e.g., to the unit interval.

Unit scaling is structured, but not objective, since it stills defines weights. Intuitively this may seem unavoidable, but it is actually possible to define a weightless selection method. Given a a partially ordered set (poset) of solutions, one objective selection method is to only allow the set of maximal elements of the poset: those that have no other solution ranking above them

in at least one dimension. We say that as a result these *non-dominating* solutions make up the highest ranking options between which an objective decision maker is indifferent (the maximal *indifference set*). *Pareto optimality* is a situation where no alternative exists that improves on at least one scalar without disadvantaging another. Clearly, the set of all Pareto optimal solutions, known as the *Pareto front* of the search space, is a subset of the maximal indifference set. Pareto fronts are approximated by genetic algorithms when the property of non domination is valued in the fitness function [DPAM00, ZLT01]. A generic method for expressing a Pareto optimality constraint in goal programs is not known [LA07].

Pareto optimality is not necessarily consistent with subjective notions of optimality. One well known drawback is that Pareto constraints say nothing about balance among objectives. On the contrary, note that a solution that maximizes one dimension, but minimizes all others (e.g., the normalized three dimensional vector [1,0,0]) is Pareto optimal. In practice, we invariably prefer the equally Pareto optimal vector [0.9,0.9,1]. Preference is given to well-balanced solutions by ranking the solutions in the Pareto-front. Again, we strive to avoid using subjective measures. A norm is a distance metric in a metric space. The most common norm in Euclidean space is probably the magnitude, as given by the Pythagorean of a vector's coordinates. The second example vector above has a Euclidean length of approximately 1.62, whereas the first only measures 1. We could use Euclidean distance, therefore, as an objective norm. Instead, we follow common practice by taking the Chebyshev norm  $\|x_i\|_\infty$ , which is the supremum of length in each individual dimension  $x_i$ :  $\max\{|x_{i1}|, |x_{i2}|, \dots, |x_{in}|\}$ . Objective is not to select the solution with the maximal Chebyshev norm – then the vectors would be indistinguishable. Instead, we use this norm to calculate the minimal distance for a given solution from an unattainable *aspiration point*  $\hat{x}$ , that combines the maxima observed for each individual dimension  $\hat{x}_i$ . This approach imposes a degree of fairness on the solution, by limiting negative outliers. The objective then becomes to minimize the maximal (or ‘worst’) deviation in any dimension. This solution is known as the min-max optimum because it "yields the “best” possible compromise solution" [Esc88]. For this reason it is also called the solution with the “least objective conflict”. This can be expressed as the achievement scalarizing function (ASF)

$$c(\vec{x}_i) = \sum_{j=1}^n |x_{ij} - \hat{x}_j|$$

Again, we assume for simplicity that all dimensions are already normalized to unit length. Now, the first vector has a Chebyshev norm of  $|1 - 1| + |0 - 0.9| +$

$|0 - 1| = 1.9$  and the second of  $|1 - 0| + |0.9 - 0.9| + |1 - 1| = 1$  and the second is preferred. The min-max optimum can be calculated over all solutions or only over the Pareto front, depending on whether optimality or balance is valued higher. The present implementation of Streamline uses the second.