OR Counterparts to AI Planning

Robert Fourer
Department of Industrial Engineering and Management Sciences
Northwestern University
2145 Sheridan Road
Evanston, Illinois 60208-3119
4er@iems.nwu.edu

Abstract

The term Planning is not used in Operations Research in the sense that is most common in Artificial Intelligence. AI Planning does have many features in common with OR scheduling, sequencing, routing, and assignment problems, however. Current approaches to solving such problems can be broadly classified into four areas: Combinatorial Optimization, Integer Programming, Constraint Programming, and Local Search. These areas have developed somewhat independently; they have characteristic strengths and weaknesses, and have been commercially developed to varying degrees.

Any comparison of the operations research (OR) and artificial intelligence (AI) approaches to planning must start by addressing a discrepancy in the key terminology. The term *AI Planning* refers to the study of a distinct class of problems for which a technical definition can be given (Nareyek 2000):

> The basic planning problem is given by an initial world description, a partial goal world description and a set of actions/operators that map a partial world description to another partial world description. A solution is a sequence of actions that leads from the initial world description to the goal world description and is called a plan. The problem can be enriched by including further aspects, like temporal or uncertainty issues, or by requiring the optimization of certain properties.

In OR and related fields such as management science and industrial engineering, "planning" is a more informal term. To the extent that it is well defined, moreover, it describes something very different from planning in AI. As an example, a recent article on OR consulting (Crowder 1997) gives the following advice:

> Scheduling involves short time periods, disaggregated data, small segments of the business, and answers operational kinds of questions of interest to people who actually do work. Planning involves longer time periods, aggregated data, larger business units, and gives solutions to strategic kinds of problems of interest to people who only talk about doing work.

The OR analogue to AI planning would thus seem to be *scheduling*. Terms such as sequencing, routing, and assignment are also widely used to describe more specialized kinds of scheduling. Planning in OR, on the other hand, would seem to be more nearly the opposite of what it is in AI!

This article thus aims to describe the OR views of scheduling and related problems, which readers may find to provide some interesting analogies to the AI view of planning. Still, it is worth keeping in mind that the analogy is imperfect at best, as terms in either field carry certain connotations absorbed from the most common applications. Thus one paradigm for AI planning could be robot control, a problem almost unknown in OR circles. On the other hand, typical examples of OR scheduling involve work assignments such as jobs to machines or crews to flights, which would appear to lie outside of the standard definitions of AI Planning. Techniques borrowed from AI have started to prove useful for these OR scheduling problems, however, and hence I will take scheduling in the OR sense as the starting point for comparisons in this article.

There is no one way of scheduling in OR. The classification that I adopt here distinguishes four main ways of approaching the task of finding a solution:

- Combinatorial Optimization
- Integer Programming
- Constraint Programming
- Local Search

Although these can be viewed as merely solution methods, any of which could be applied to any scheduling problem, in practice each tends to be associated with certain scheduling applications and modeling techniques.

Combinatorial Optimization

Virtually any scheduling task can be viewed as a combinatorial (or discrete) optimization problem to which established complexity theory may be applied. Thus there is a huge body of complexity results for scheduling, addressing the minimization of makespan, total tardiness, weighted completion time, and other objectives under different combinations of processor numbers and times, release dates, precedences, preemption rules, due dates, and many other characteristics. The variety of combinations has been sufficient to support specialized terminology, classification schemes, and

even software for keeping track of known results (Lageweg *et al*. 1982, Brucker and Knust 2000). The technical literature on the subject began as theoretical computer science but has long been based mainly in OR-related publications.

As in other areas of combinatorial optimization, the core of the effort has been to divide problems into those that can be solved with polynomially-bounded work and those that can be proved to be NP-hard. There is also a concern with reducing the size of bounds on the worst-case effort of optimal algorithms (for polynomial problems) and heuristics (for HP-hard problems).

Problems of practical interest tend to involve many kinds of complications specific to the situation at hand. As a result, practical scheduling problems often fail to precisely match any of those that have been studied mathematically. Practical combinatorial optimization for scheduling tends to be more of an empirical science, based on the creation of iterative schemes that work well for given applications. Evaluation of these schemes naturally focuses more on their average behavior than on the worst case, while the role of theory is to establish which subproblems are amenable to being solved or approximated efficiently. Little can be said about these schemes in general terms, however, as their design and implementation tends to be highly problem-specific.

Integer Programming

Virtually any combinatorial optimization problem — hence any scheduling problem — can be written as a minimization or maximization of integer decision variables, subject to linear equalities and inequalities in the variables. Problems of this latter sort are known as *integer programs,* or as *mixed integer programs* or *MIPs* when additional continuous decision variables are also employed.

Branch-and-Bound

Powerful general-purpose MIP solvers have been refined over several decades, and are capable of solving problems in thousands of integer variables. They employ a kind of divide-and-conquer strategy that builds a tree of progressively more restricted problems, terminating with specific solutions at the leaf nodes. The potentially exponential work of searching the whole tree is greatly reduced through analysis of the relatively easy *linear program* that results upon relaxing the integrality requirement for each node's restricted problem. (The linear program's solution can be further speeded up by starting from the solution of its parent's relaxation.) The relaxation's solution at a node sometimes turns out to be all-integer, and in other cases provides a bound that is worse than some all-integer solution already found; in either case, the tree can be "pruned" at that point.

A *branch-and-bound* strategy of this kind can be refined through heuristic node and branch selection strategies for building the tree and through simplifications of the restricted problems at the nodes. Often these operations can make further use of information provided by the solution of the linear programming relaxation. Upper and lower bounds on the optimal objective value can also be maintained, and the gap between them shrinks as the search proceeds.

The integer programming approach has the appeal of using off-the-shelf software for formulating and solving. Consider for example the just-in-time sequencing problem (Jordan and Drexl 1995) in which jobs $j \in \mathcal{J}$ have processing times p_j and must be completed by due dates d_j, but incur an earliness penalty e_j for each day that they are completed early. Processing job i immediately after job j incurs both a setup penalty s_{ij} and a setup delay t_{ij}. The problem of choosing a job order that minimizes total penalty can be formulated as a mixed integer program in terms of two groups of variables:

- Continuous variables U_j are the completion times of the jobs $j \in \mathcal{J}$.
- Integer variables V_{ij} are 1 if job j immediately follows job i, and zero otherwise, for all $i \in \mathcal{J}$ and $j \in \mathcal{J}$ with $i \neq j$.

The objective and constraints can then be written as:

$$
\begin{array}{lll}
\text{Min} & \sum_{j\in\mathcal{J}} e_j(d_j - U_j) + & \\
& \sum_{i\in\mathcal{J}} \sum_{j\in\mathcal{J}:i\neq j} s_{ij}V_{ij} & \\
\text{Subj to} & \sum_{i\in\mathcal{J}} V_{ij} = 1, & j \in \mathcal{J} \\
& \sum_{i\in\mathcal{J}} V_{ji} = 1, & j \in \mathcal{J} \\
& U_i + t_{ij} + p_j \leq U_j + B(1 - V_{ij}), & \\
& & i \in \mathcal{J}, j \in \mathcal{J}\text{: } i \neq j \\
& 0 \leq U_j \leq d_j, & j \in \mathcal{J} \\
& V_{ij} \in \{0, 1\}, & i \in \mathcal{J}, j \in \mathcal{J}\text{: } i \neq j
\end{array}
$$

The constant B must be chosen sufficiently large to make the constraint in which it appears non-binding when $V_{ij} = 0$. (We should also be more precise in the formulation's handling of the beginning and end of the job sequence, but in the interest of simplicity we have omitted the details here and in subsequent related examples.)

The use of integer programming has been further promoted by the availability of modeling languages that encourage experimentation with alternative formulations. Currently the most popular among these are the *algebraic modeling languages* (Kuip 1993) that permit MIP formulations to be expressed in a common and natural mathematical form. Here is the model above, for example, in the AMPL language (Fourer, Gay and Kernighan 1990, 1993):

```
set J;

param d {J};
param e {J};
param p {J};

param s {J,J};
param t {J,J};

var U {j in J} >= 0, <= d[j];
var V {i in J, j in J: i <> j} binary;

param B;

minimize Total_Penalty:
   sum {j in J} e[j] * (d[j] - U[j]) +
   sum {i in J}
      sum {j in J: i <> j} s[i,j] * V[i,j];
```

```
subj to OneBefore {j in J}:
   sum {i in J} V[i,j] = 1;

subj to OneAfter {j in J}:
   sum {i in J} V[j,i] = 1;

subj to NoOverlap {i in J, j in J: i <> j}:
   U[i] + t[i,j] + p[j]
     <= U[j] + B * (1 - V[i,j]);
```

More mnemonic names for parameters and variables could be used if desired. A particular instance of this model would be created by processing it along with specific values for the declared set and parameters.

The key feature of modeling languages is their ability to declare models and data without giving any procedural description of how a solution is to be computed. Thus the optimization can be delegated to any of a variety of MIP solvers. Modeling languages are typically implemented within computing environments that provide support for managing models, data, and solutions. Procedural extensions for scripting series of solves and more elaborate iterative schemes are also available, but the underlying description of the models remains declarative.

The chief drawback of the integer programming approach is also seen in this example. Integer programming usually requires the construction of a formulation that is significantly different from the scheduling problem's original statement and that involves large numbers of additional variables and constraints. Worse yet, the performance of even the most sophisticated branch-and-bound implementations is highly sensitive to the formulation. Many simple and concise formulations turn out to require too large a search tree to be practical.

The construction of "good" formulations is something of an art, often involving the addition of complex classes of redundant constraints that make the formulation more suitable for branch-and-bound, though they leave the optimal value of the integer program unchanged. Constraints that reduce the number of optimal solutions by "breaking symmetries" are often effective in curtailing fruitless searching. Other added constraints lead to tighter bounds from the relaxations, permitting more of the search tree to be pruned. These latter constraints are known as *cuts* because they cut off some of the fractional solutions of the relaxed problems. In addition to the problem-specific cuts introduced by clever modelers, good branch-and-bound implementations automatically generate cuts of various kinds.

Branch-and-Price

Even those integer programming formulations that would seem to have an impossibly large number of variables can be handled if the variables have some regular pattern.

As an example, consider the problem of scheduling workers over a series of T days so that at least d_t worker-hours are scheduled on each day t. This could be formulated in terms of variables X_{it} representing the number of hours that person i works on day t, with constraints to ensure that sufficient worker-hours are scheduled each day and that each worker's schedule meets certain contract requirements. If the requirements are the same for all workers, however, then this leads to a highly symmetric formulation that is hard to solve.

Instead, suppose we begin by listing all contractually acceptable work plans. We can let a_{jt}, say, represent the number of hours that a person assigned to work plan j will put in on day t. Then the variables are simply the numbers of people Y_j that are assigned to follow work plan j. If the total pay for work plan j is c_j, then the scheduler's problem can be written as

$$\begin{array}{ll} \text{Min} & \sum_{j\in\mathcal{J}} c_j Y_j \\ \text{Subj to} & \sum_{j\in\mathcal{J}} a_{jt} Y_j \geq d_t, \qquad t = 1,\ldots,T \\ & Y_j \in \{0,1,2,3,\ldots\}, \quad j \in \mathcal{J} \end{array}$$

Unless the contract rules are highly restrictive, however, the size of the set $\mathcal{J}$ is likely to grow exponentially with the number of days, producing far too many variables to permit this formulation to be used directly.

To deal with this situation, the branch-and-bound process can be initiated with only a subset of work plans. Additional useful work plans can be generated as the search proceeds, by solving a pattern-generating subproblem at some of the nodes; in fact this subproblem can itself be formulated as a combinatorial optimization problem, in which the variables Z_t are the numbers of hours to be worked on day t in the plan to be added, and the constraints enforce the contract rules. The objective for this subproblem is properly expressed in terms of the "dual prices" generated in solving the restricted linear programming relaxations at the branch-and-bound nodes, leading the overall scheme to be known as *branch-and-price*. Space does not permit a discussion of the details, which become quite complex for practical situations such as the scheduling of airline crews.

Branch-and-Cut

A similar approach can be taken in cases where the number of constraints is too large to handle.

An example is provided by the simple routing problem known as the "traveling salesman" problem. Given m cities and distances d_{ij} between all pairs of cities i and j, a "tour" of all the cities is desired such that the total distance traveled is as small as possible. We can define variables X_{ij}, $i < j$, to be 1 if the tour goes directly between cities i and j, and 0 otherwise. Then the total cost is the sum of the terms $d_{ij}X_{ij}$; for the constraints we need only say that the tour goes directly between each city and exactly two other cities, and that the solution cannot contain a "subtour" on any subset of cities:

$$\begin{array}{ll} \text{Min} & \sum_{i=1}^{m}\sum_{j=i+1}^{m} d_{ij}X_{ij} \\ \text{Subj to} & \sum_{i=1}^{k-1} X_{ik} + \sum_{j=k+1}^{m} X_{kj} = 2, \\ & \quad k = 1,\ldots,m \\ & \sum_{i\in\mathcal{S}}\sum_{j\in\mathcal{S}:i\neq j} X_{ij} \leq |\mathcal{S}| - 1, \\ & \quad \mathcal{S} \subset \{1,\ldots,m\} \text{ with } 3 \leq |\mathcal{S}| \leq m/2 \end{array}$$

The idea of the "subtour elimination" constraints is that a solution cannot contain a subtour on some subset $\mathcal{S}$ of cities if it involves links between fewer than $|\mathcal{S}|$ pairs of cities in $\mathcal{S}$. The number of these constraints clearly grows exponentially

with the size of $\mathcal{S}$, however, and hence is impossibly large for all but the smallest problems.

Again, the problem of exponential growth in the formulation can be handled by a strategy of incremental generation — this time, of constraints. Basically, we start by attempting to solve the integer program without the subtour elimination constraints. But whenever we encounter a solution that contains a subtour, we add the constraint that eliminates that subtour. Experience suggests that the number of constraints that need to be added in this way grows at a much more reasonable rate than the total number of subtour elimination constraints.

The same ideas can be used to incrementally generate cuts that are redundant for the integer programming formulation but that are advantageous to branch-and-bound as explained previously. Hence this approach is known as *branch-and-cut*. Studies of the traveling salesman problem have uncovered intricate cut structures that permit branch-and-cut to solve problems in thousands of cities (Applegate, Bixby, Chvatal and Cook, 1998). For practical purposes, however, the interest is in more complicated routing problems to which this approach can be applied. If we replace "cities" by "jobs" and interpret d_{ij} as the setup time between jobs, moreover, then the same formulation is seen to apply to some kinds of machine scheduling.

Constraint Programming

Enumerative search methods were developed independently in the AI community, initially for solving problems in which the variables take only true-false values and are connected into constraints by logical operators (such as "or" and "not"). The extent of the search can be reduced in this case by applying certain methods of "resolution" that can be viewed as analogues to cut generation in branch-and-bound. (There is no analogue for the idea of a bound, however.) Extensions of this approach to more complex kinds of variables and constraints has produced a variety of *constraint programming* languages and systems.

From an OR standpoint, the significance of constraint programming lies in permitting enumerative search to be applied to more natural formulations of combinatorial problems. A typical constraint programming formulation of the preceding sequencing model, for example, would replace the zero-one variables V_{ij} by "job-valued" variables W_k, such that W_k represents the job that is kth in the job sequence:

$$\begin{aligned}
\text{Min} \quad & \textstyle\sum_{j\in\mathcal{J}} e_j(d_j - U_j) + \sum_{k=1}^{|\mathcal{J}|} s_{W_k,W_{k+1}} \\
\text{Subj to} \quad & \textit{all-different}_{k\in\mathcal{J}}\, W_k \\
& U_{W_k} = \min(d_{W_k}, U_{W_{k+1}} - p_{W_{k+1}} - t_{W_k,W_{k+1}}) \\
& \qquad\qquad k \in 1,\ldots,|\mathcal{J}| \\
& W_k \in \mathcal{J}, \qquad\qquad k \in 1,\ldots,|\mathcal{J}|
\end{aligned}$$

An expression such as U_{W_k} is read directly as "the completion time of the kth job." Such "variable in subscript" expressions permit the $|\mathcal{J}|^2$ variables V_{ij} to be replaced by $|\mathcal{J}|$ variables W_k, each having a larger domain. Linearity of the expressions is not required, permitting the constraint that relates production, setup, and completion times to be changed from an inequality (in the integer programming formulation) to an equality; this constraint now fully determines the U_j variables once the W_k variables have been enumerated. Finally, the *all-different* constraint replaces $2|\mathcal{J}|$ individual constraints from the integer program.

A constraint programming solver builds a search tree for this problem directly, with branches restricting the solution to different cases as appropriate. Each restriction is "propagated" to further reduce the domains of the variables in descendant nodes; this is the main device for keeping the size of the search tree manageable. With a judicious choice of the order in which variables are restricted to yield nodes and branches, this approach can solve certain difficult scheduling problems significantly faster than branch-and-bound applied to an integer programming formulation.

The initial idea of constraint programming was to embed constraints into a full-featured programming language. More recently, however, the idea of an algebraic modeling language has been extended to formulations like the one above (Fourer 1998, Van Hentenryck 1999). Indeed, there has been considerable cross-fertilization between integer programming and constraint programming in recent years, at the levels of both solving and modeling. Chances are that they will become less distinct over time.

Local Search

The enumerative search methods of integer and constraint programming are "global" in the sense that they are guaranteed to find the best of all possible schedules, sequences, or assignments, if only they can be run long enough. The major alternative is a "local" approach that finds progressively improved schedules by making incremental improvements of some kind. Although local search cannot offer strong guarantees of optimality, it has compensating advantages in various practical situations.

In its simplest form, local search requires an admissible solution (what would be called a "feasible" solution in integer programming) to start from, and a rule that defines for each such solution a collection of "neighboring" solutions that are also admissible. A step of the search then involves moving from one solution to a neighboring solution that has a better objective value. The search continues until it reaches a solution whose objective value is at least as good as any of its neighbors' values; at that point it has found a "locally optimal" solution.

In many sequencing problems, for example, jobs may miss their due dates, at the cost of incurring some penalty to the objective. (In fact the entire objective may be to minimize total tardiness.) Then any sequence of jobs may be a valid solution. The neighbors of a solution might be all those obtained by exchanging some pair of jobs in the sequence, or by moving some one job to a different place in the sequence. A similar analysis can be applied to the traveling salesman problem, but in that case it might make more sense to generate neighbors by replacing any two links $i_1 \to i_2$ and $j_1 \to j_2$ with $i_1 \to j_1$ and $i_2 \to j_2$. In general, we would like the neighborhood rule to generate neighbors that have a good chance of giving better objective values.

Unfortunately, most scheduling problems have huge numbers of locally optimal solutions that are far from the global optimum. Even if local search is enhanced by choosing at each step the neighbor with the best objective value — a "greedy" algorithm — and by trying many starting solutions, results are often disappointing.

Practical local search methods must thus incorporate some device for getting away from local minimums. Indeed, local search methods are categorized according to the devices they use. *Simulated annealing* (van Laarhoven and Aarts 1987) allows occasional steps to solutions with worse objective values, according to a probability that is gradually reduced. *Tabu search* (Glover and Laguna 1997) also permits steps to inferior solutions, but promotes eventual progress by maintaining a list of recent solutions that may not be revisited. *Genetic algorithms* (Liepins and Hilliard, 1989) start with a set of valid solutions; at each step new solutions are constructed by combining parts from existing pairs of solutions, and some existing solutions are discarded, according to criteria that tend to promote the survival of the best solutions. Other local search methods of current interest include ant colony optimization, differential evolution, immune system methods, memetic algorithms, and scatter search (Corne, Dorigo and Glover 1999). Numerous refinements have enabled such methods to adapt more readily to varying circumstances and to deal with constraints.

Although a kind of asymptotic global optimality can be assured for some local search methods, as a practical matter these methods are used where circumstances do not require or permit a guarantee of optimality or a bound on the current solution's distance from optimality. They are particularly attractive where the form of the objective function or the set of valid solutions does not lend itself to the more analytical approaches previously described. Local search can also provide a way of computing good schedules quickly and with limited investment in sophisticated software. If a good starting solution is known, moreover, then the effort of local search focuses specifically on improving it.

Local search methods represent general approaches more than specific algorithms, however. Thus they typically require some amount of programming to be set up for specific problems. The construction of a good neighborhood rule may also be difficult. Individual steps do tend to be fast, but the number of steps can be very large, particularly as there are no bounds generated to give a stopping rule. Performance may be sensitive to the settings of parameters that govern, say, the composition of a tabu search list or the choice of solutions to be combined or discarded in a genetic algorithm; repeated trial runs may be necessary to develop good settings.

Availability

There is a long tradition of commercial integer programming software. To provide good performance on a range of problems, effective branch-and-bound implementations must offer a variety of carefully tuned options for preprocessing, node and branch selection, cut generation, and other operations. The best codes are thus developed over a period of years and are not easily reproduced; as a result, they tend to be expensive (Fourer 1999). Practical constraint programming software has similar characteristics, and appears to be developing commercially in a similar way.

All of the widely used modeling languages and systems are also commercial in nature. Typically they are usable for planning as well as scheduling and for many other optimization problems, via a diversity of features that have been developed over a long time. New languages are thus not readily introduced, and again the software tends to be expensive.

Combinatorial scheduling algorithms and scheduling applications of local search tend to be more specialized, but also more readily available as research codes at little or no cost. There is no one best source of information for these categories, but the software is not hard to track down by means of a Web search.

Acknowledgments

This work has been supported in part by National Science Foundation grants DMI-9414487 and DMI-9800077 to Northwestern University.

References

D. Applegate, R. Bixby, V. Chvatal and W. Cook, 1998. On the Solution of Traveling Salesman Problems, *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung*, International Congress of Mathematicians, 645–656; `www.keck.caam.rice.edu/tsp/index.html`.

P. Brucker and S. Knust, 2000. Complexity Results for Scheduling Problems, `www.mathematik.uni-osnabrueck.de/research/OR/class/`.

D. Corne, M. Dorigo and F. Glover, eds., 1999. *New Ideas in Optimization*, McGraw-Hill.

H.P. Crowder, 1997. Seven Helpful Hints for OR/MS Consultants, *OR/MS Today* **24,** 1 (February).

R. Fourer, 1998. Extending a General-Purpose Algebraic Modeling Language to Combinatorial Optimization: A Logic Programming Approach. In D.L. Woodruff, ed., *Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search: Interfaces in Computer Science and Operations Research*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 31–74.

R. Fourer, 1999. Software Survey: Linear Programming, *OR/MS Today* **26,** 4 (August) 64–65; `lionhrtpub.com/orms/orms-8-99/survey.html`.

R. Fourer, D.M. Gay and B.W. Kernighan, 1990. A Modeling Language for Mathematical Programming, *Management Science* **36,** 519–554.

R. Fourer, D.M. Gay and B.W. Kernighan, 1993. *AMPL: A Modeling Language for Mathematical Programming*, Duxbury Press, Pacific Grove, CA.

F. Glover and M. Laguna, 1997. *Tabu Search.* Kluwer Academic Publishers.

C. Jordan and A. Drexl, 1995. A Comparison of Constraint and Mixed-Integer Programming Solvers for Batch

Sequencing with Sequence-Dependent Setups, *ORSA Journal on Computing* **7,** 160–165.

C.A.C. Kuip, 1993. Algebraic Languages for Mathematical Programming, *European Journal of Operational Research* **67,** 25–51.

B.J. Lageweg, J.K. Lenstra, E.L. Lawler and A.H.G. Rinnooy Kan, 1982. Computer-Aided Complexity Classification of Combinatorial Problems, *Communications of the ACM* **25,** 817–822.

G.E. Liepins and M.R. Hilliard, 1989. Genetic Algorithms: Foundations and Applications, *Annals of Operations Research* **21,** 31–57.

A. Nareyek, 2000. EXCALIBUR: Adaptive Constraint-Based Agents in Artificial Environments — Planning, `www.ai-center.com/projects/excalibur/documentation/intro/planning/`.

P. van Hentenryck, *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA.

P.J.M. van Laarhoven and E.H.L. Aarts, 1987. *Simulated Annealing: Theory and Applications*. D. Reidel, Norwell, MA.