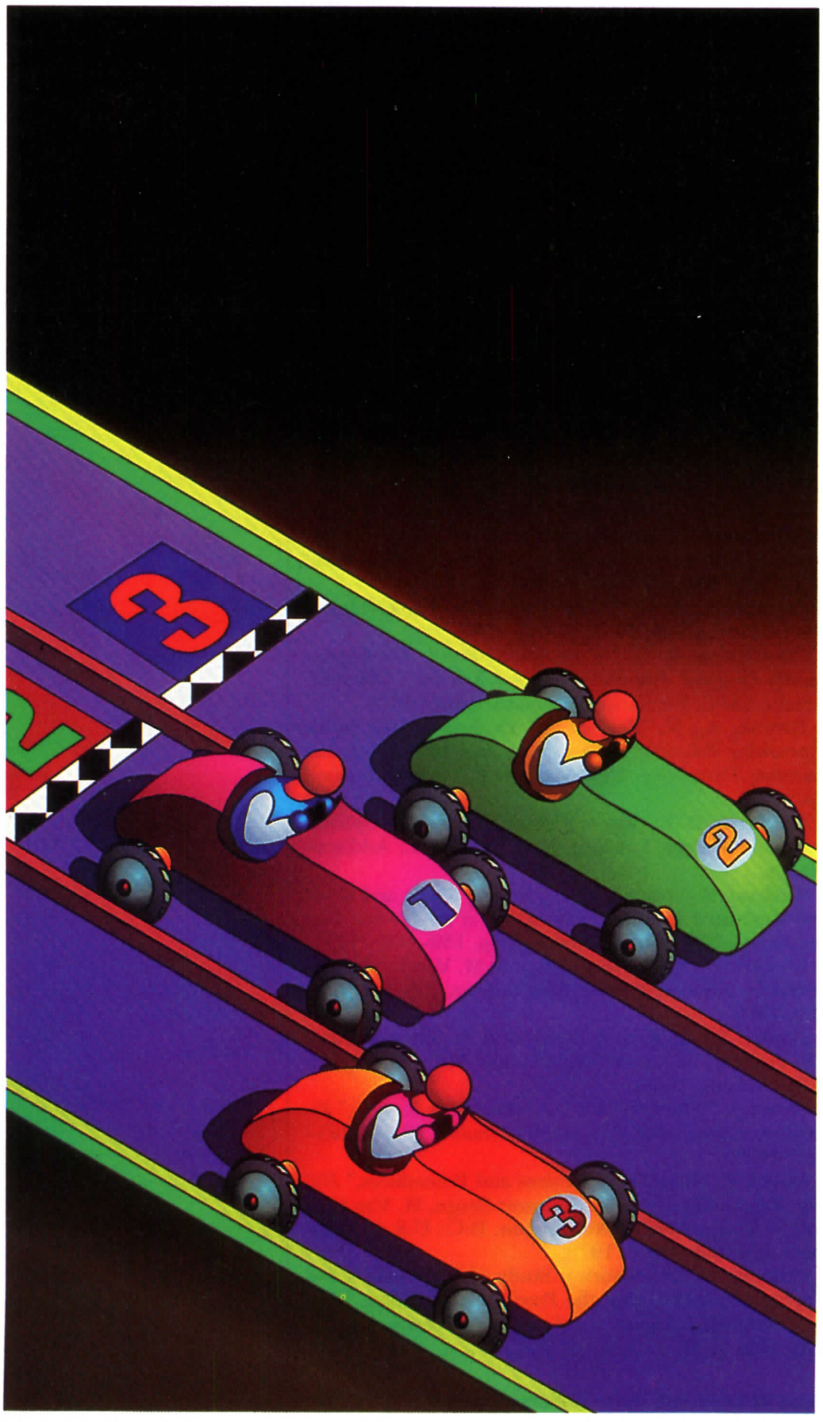
Sequencing problems usually mean a computer must search the entire state space for a solution, which is a time-consuming chore—intelligent heuristics can help



Many problems that cannot be solved algorithmically are search problems, and finding a solution that satisfies the system requirements necessitates searching through a large number of possible solutions. For example, in a manufacturing environment, management must accommodate constraints on start and completion time and schedule particular operations for particular machines. Often, on even the fastest computers, all possible assignments of operations to machines cannot be tested in a reasonable amount of time. In such cases, intelligent heuristics must be developed to prune the search space. We will explore a heuristic for reducing the search space for designing schedules that assign specific operations to specific operators. The specific application we designed with the algorithm and heuristic was a round-robin schedule for Cub Scout Pinewood Derbies. However, this heuristic can certainly be applied to manufacturing, scheduling commercial air traffic, and scheduling competitive events.

Since the 1950s, more than 80 million Cub Scouts have built five-ounce wooden cars and raced them in Pinewood Derbies. Pinewood Derbies have traditionally consisted of a series of elimination races where only the winner from each heat proceeds to the next round. This set up pleases Scouts with fast cars, but for the unlucky majority it means a single race, waiting for the awards to be announced, and then going home.

The best way to explain our Cub Scout problem is to first show a solution. Table 1 shows a schedule for a 15-car, six-round race. Each car is identified with a letter, *A* through *O*. Each car runs six times, twice in each lane (most tracks have three lanes), and only runs against any particular competitor once. This schedule is the end result of our project. Although it looks simple, it was not easy to derive.

In our paradigm, we changed the race format for our Cub Scout pack to a round robin (see Table 1). The objective was to let each Scout race more often and throughout the whole event. We decided to use six

Reducing State Space Search Time: *Scheduling in the Classic AI Challenge*

By A. Terry Bahill and William J. Karnavas

rounds because that would give each car two runs in each lane and still keep the whole event reasonably short. Switching from an elimination tournament to a round robin produced two side benefits: The Scouts raced more often, and lane biases were eliminated because each car ran in each lane the same number of times.

The next task was to derive schedules. The schedules had two constraints: Each car had to run twice in each of the three lanes, and no two cars should run against each other more than once. We knew from past Pinewood Derby experiences that schedules for nine to 39 cars were required. We only created schedules for races where the number of cars was a multiple of three. When the number of cars was not a multiple of three, we merely left some lanes empty.

In 1989, we created the first schedules by hand. Each car was assigned a letter, and the first round was assigned in alphabetical order. The slight deviation from alphabetical order shown in Table 1 (interchanging *N*

ARTWORK: BARTON STABLER

and *J*) ensured that all heats would have at least two cars, even if no cars were labeled *N* and *O*. Subsequent rounds were filled by rearranging the first round using rotations, then trial and error. Our 15-car, hand-derived schedule supposedly satisfied both constraints, but later we found it satisfied neither. We therefore decided to give up on making the schedules by hand and investigated computer algorithms.

RELATED STUDIES

Kirkman's Schoolgirls Problem is similar to our Pinewood Derby problem. According to Rouse Ball and Coxeter's *Mathematical Recreations & Essays*, in 1850 T. P. Kirkman said: "A school mistress took her 15 girls for a daily walk; the girls were arranged in five rows of three girls each so that each girl had two companions." The task was then to "Plan the walk for seven consecutive days so that no girl walks with any of her classmates in any triplet more than once."

There are several clever tricks for finding solutions to this problem. One simple graphical technique works if there are n girls, such that $n = 24m + 3$ or $n = 24m + 9$, where m is a member of the set $\{0, 1, 2, 3, \ldots\}$. Solutions exist for all values of n where n is an odd multiple of three, but they involve trial and error and are much harder to implement.

Kirkman's problem is bigger than ours because the girls go on seven walks, whereas our scouts only race six times. However, our problem is harder because of the lane restriction: Each scout must race in each lane exactly twice. We could not adapt any of the Kirkman Schoolgirls algorithms to our problem. Checking the usefulness of the Kirkman solutions for the Pinewood Derby was a large combinatorial problem that we decided was not a beneficial endeavor. However, we applied our algorithm and heuristic to the schoolgirl problem on a 486 PC, and so far we found 368 of the 845 solutions.

Montgomery showed how AI heuristics could be used to improve state-space searches. The first problem he presented, the Magic Star, took his heuristic two hours to find 959 solutions. Our algorithm and heuristic ran for 12 minutes (including validation of each solution) and found 960 solutions. His second example was the Who Lives on First Problem: "Five houses are on a certain street, each of a different color, and inhabited by eccentric old men of different nationalities, different pets, drinks, and smokes. The following information is known:

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. The Ukrainian drinks tea.
°
°
16. One of the men owns a zebra.

As you might expect, the problem is to dis-

The Permutation-

This algorithm is best explained by the following small example. The right column shows the 24 permutations of the letters *A*, *B*, *C*, and *D*, and the left side shows the operations needed to produce each of these permutations.

		Position 3 2 1 0
Step-0:	Initial sequence	A B C D
Step-1:	Swap 1 and 0 of sequence above	A B D C
Step-2:	Swap 2 and 0 of initial sequence	A D C B
Step-3:	Swap 1 and 0 of sequence above	A D B C
Step-4:	Swap 2 and 1 of initial sequence	A C B D
Step-5:	Swap 1 and 0 of sequence above	A C D B
Step-6:	Swap 3 and 0 of initial sequence	D B C A
Step-7:	Swap 1 and 0 of sequence above	D B A C
Step-8:	Swap 2 and 0 in sequence of Step-6	D A C B
Step-9:	Swap 1 and 0 of sequence above	D A B C
Step-10:	Swap 2 and 1 in sequence of Step-6	D C B A
Step-11:	Swap 1 and 0 of sequence above	D C A B
Step-12:	Swap 3 and 1 of initial sequence	C B A D
Step-13:	Swap 1 and 0 of sequence above	C B D A
Step-14:	Swap 2 and 0 in sequence of Step-12	C D A B
Step-15:	Swap 1 and 0 of sequence above	C D B A
Step-16:	Swap 2 and 1 in sequence of Step-12	C A B D
Step-17:	Swap 1 and 0 of sequence above	C A D B
Step-18:	Swap 3 and 2 of initial sequence	B A C D
Step-19:	Swap 1 and 0 of sequence above	B A D C
Step-20:	Swap 2 and 0 in sequence of Step-18	B D C A
Step-21:	Swap 1 and 0 of sequence above	B D A C
Step-22:	Swap 2 and 1 in sequence of Step-18	B C A D
Step-23:	Swap 1 and 0 of sequence above	B C D A

The steps can be assigned a level, *S*, based on the highest position they swap. Steps-6, 12, and 18 all swap position three with lower positions and thus are called level-3 swaps. Steps-2 and 4 (and all their repetitions) swap position two with lower positions and thus are called level-2 swaps. Step-1 (and all its repetitions) swaps position one with its lower position, zero, so it is the level-1 swap. A level-*S* swap changes the character at position *S* with lower positions starting at zero and ending with *S*-1: Therefore, it can make *S* swaps before the sequence repeats.

A level-*S* swap changes characters in its working sequence with the last sequence output by a higher level (the initial sequence is the highest level). So a level-1 swap is always done to the previous output. A level-2 swap is done to the last level-3 or higher output. The level-3 swap is always carried out on the initial sequence (for the four-character example).

To implement the algorithm, all that is needed is to hold a count and working sequence for each level. Each time a level exhausts all the permutations it can make, the higher level is invoked. The higher level makes its swap and then

cover which man lives in which house, what the color of each house is, and what each man's pet, drink, and smoke is."

There are 25 billion possibilities, but by using an intelligent heuristic, he was able to solve the problem on an IBM PC in two hours. Our program solved this problem in two minutes. However, we are not claiming that our program is better because it is faster: Although we used the same computer, we used a different language. Our program is more general since we used the exact same code in each case to generate the permutations and only modified the functions that check constraints.

Generating Algorithm

resets the counts of the lower levels and sets their working sequence to its present output. This lends itself easily to a recursive implementation but can also be implemented iteratively. For speed, we implemented it as an iterative function that returns the next permutation of the sequence and only required a small record be kept from call to call. The following Pascal program shows this implementation:

```
Program Permutations( Output );
Const
   N = 4;
   NFact = 24;
Type
   PermStr = Array[0..N-1] of Char;
Var
   Counts: Array[1..N] of Integer;
   WorkSequence: Array[1..N] of PermStr;
   PrintStr: PermStr;
   i: Integer;

Procedure Permutation( Level: Integer; Var OutStr: PermStr );
Var C: Char;
Begin
   If Counts[Level] >= 0 Then Begin
      OutStr := WorkSequence[Level];
      C := OutStr[Level];   (* swap 2 characters *)
      OutStr[Level] := OutStr[Counts[Level]];
      OutStr[Counts[Level]] := C;
      Counts[Level] := Counts[Level] - 1;
   End Else Begin
      Permutation(Level+1, OutStr);
      Counts[Level] := Level-1;
      WorkSequence[Level] := OutStr
   End;
End;

(* Main program *)
Begin
      (* set up variables used by Permutation *)
   For i := 1 to N-1 Do
      Counts[i] := -1;
   Counts[N] := N;
   WorkSequence[N] := 'ABCD';
      (* find and print the permutations of a sequence *)
   For i := 1 to NFact Do Begin
      Permutation( 1, PrintStr );
      Writeln( PrintStr )
   End
End.
```

RANDOM TRIALS SOLUTION

We will use the following terminology to describe Pinewood Derbies. We will refer to each set of cars running down the track at the same time as a "heat." The number of heats necessary for every car in the division to run once constitutes a "round." A set number of rounds constitutes a "divisional race." Thus, for 15 cars to run six times each, the divisional race consists of six rounds of five heats each, as shown in Table 1. Finally, several divisional races constitutes a "derby." Typically the divisions are aggregated by age, with four or five divisions being common in a Cub Scout pack.

Our first computerized scheduling system was a simple program that randomly ordered cars in races. It successfully produced round-robin schedules for 18 or more cars. The first round was assigned in alphabetical order as was done by hand. The program successively filled the slots in each heat of each round using a random number generator to assign a car. If the car had been in the round already, or violated either constraint, it was rejected, and the next car in alphabetical order was tried. If all cars were tried in a slot without success, the partial round was discarded and restarted. If the number of restarts for a round exceeded a threshold (66 was arbitrarily used), the program cleared all assignments from round two on and started again. Within minutes the program produced schedules for divisional races of 18 or more cars. However, the 15-car, six-round divisional race remained unsolved, even after letting the program churn for more than 24 hours.

Why did the random assignment program fail for 15 cars? By the calculations previously mentioned, there was no logical reason why the 15-car, six-round schedule could not be found. Our program worked well for 18 and more cars, so there was no reason to assume a programming error. Yet, it ran for over a day without finding a solution. Because it was a random guesser, there was no way to know if it would ever terminate. This situation created the impetus for us to seek out deterministic methods.

DETERMINISTIC METHODS

In an attempt to find a correct 15-car, six-round schedule, we explored two determin-

	LANE 1	LANE 2	LANE 3
		ROUND 1	
HEAT 1	A	B	C
HEAT 2	D	E	F
HEAT 3	G	H	I
HEAT 4	N	K	L
HEAT 5	M	J	O
		ROUND 2	
HEAT 1	C	F	G
HEAT 2	D	B	H
HEAT 3	A	N	M
HEAT 4	E	K	J
HEAT 5	I	L	O
		ROUND 3	
HEAT 1	L	J	C
HEAT 2	E	G	N
HEAT 3	H	F	K
HEAT 4	I	M	B
HEAT 5	O	A	D
		ROUND 4	
HEAT 1	M	C	D
HEAT 2	B	E	L
HEAT 3	G	A	K
HEAT 4	N	O	H
HEAT 5	F	I	J
		ROUND 5	
HEAT 1	J	D	G
HEAT 2	F	L	A
HEAT 3	C	I	N
HEAT 4	H	M	E
HEAT 5	K	O	B
		ROUND 6	
HEAT 1	J	H	A
HEAT 2	B	N	F
HEAT 3	K	D	I
HEAT 4	L	G	M
HEAT 5	O	C	E

TABLE 1. A 15-car, six-round race table.

istic methods for finding a solution or proving that none exists. One was a heuristic search of the 15 factorial (15!) permutations of cars that make up each round. This process involved finding a fast algorithm for generating the permutations of a sequence and checking each sequence for constraint compliance. The other method was an integer programming formulation that we will not discuss.

First, we considered the brute force approach of trying all potential schedules. The plan was simply to generate all 15! permutations of the ordering of the cars in each round and then check each against all previously accepted rounds for lane and competitor conflicts until we could find a set of six.

In this approach, we encountered two problems. First, we could not test all 15! permutations (about 10^{12}) per round in any reasonable time period. The second problem was our lack of an appropriate algorithm for generating all the permutations of a sequence. This problem was much more pressing since it prevented even attempting an exhaustive search. All permutation algorithms we had seen generated the whole set of permutations of a sequence in one shot: We did not have the resources to store all these sequences. The brute force approach was only feasible if each sequence was generated, tested, and then accepted and saved or rejected and discarded. Therefore, we developed a new recursive algorithm that can be used to generate one permutation at a time.

PERMUTATION GENERATING

The permutation-generating algorithm works by breaking the problem of generating the permutations of a sequence of n characters into the problem of generating the permutations of a sequence of $n - 1$ characters, n times. This continues until n is two, where the permutations are the sequence and the sequence reversed. Thus, the permutations of *AB* are *AB* and *BA*. To generate the permutations of *ABC*, first find the two permutations of *AB* and append *C*, then find the two permutations of *AC* and append *B*, and finally find the two permutations of *CB* and append *A*. To generate the permutations of the sequence *ABCD*, you must find the six permutations of *ABC* and append *D*, then find the six permutations of *ABD* and append *C*, and so on for *ADC* and *B*, and finally *DBC* and *A*. This example is shown in the sidebar.

This algorithm can be implemented as a recursive function that returns one permutation each time it is called. Between function calls, a record stores a count of how far it has progressed and the sequence of characters being permuted for each level of recursion. A simple Pascal version is also given in the sidebar.

Using this algorithm, we set out to find a schedule for a 15-car, six-round divisional race; it ran for four weeks on an AT&T 3B2/400 without termination or solution. To see the probable cause of this failure, compare the first round of the 15-car schedule *(ABC DEF GHI JKL MNO)* to the first permutation to be tried for the second round *(ABC DEF GHI JKL MON)*: Only *N* and *O* have changed places. Since the algorithm modifies the sequence like a counter starting in the "least significant" position (on the right end), it will require 13! permutation calls and constraint checks before *B* is changed, which is necessary to prevent *A* and *B* from racing against each other.

This led to our first heuristic rule: Find the constraint conflict in the highest position and perform a "super-permutation" at that position. A super-permutation consists of all the less-significant individual permutations needed to cause a particular character in the sequence to change. A modification of the permutation procedure implemented the super-permutation in an extremely efficient manner. By forcing all permutations to the right of a particular position to think that they are done, the sequence can permute to a new character in another position in one jump. A super-permutation requires no more work than a normal permutation and is as simple as properly setting counters of the less-significant positions in the permutation generator.

We modified the search program to implement the heuristic and used the new ability of the permutation procedure. This process involved redefining the constraint checking functions to return the position rather than the presence of conflicts. This way, the high position conflicts could be resolved in a few permutation cycles, eliminating *(p-1)!* needless permutations and checks. This made large problems such as a 39-car, six-round race feasible for systematic search, even though 39! is far too large to exhaustively search.

Our systematic search started with the first round set to alphabetic order. Then we did permutations using the heuristic until a valid round was found. We saved it and searched on from there for the next round. This process made the search faster, and we saw no reason to start each round from scratch and recheck a previously searched region of the state space.

This heuristic search program was run on large schedules and produced rapid results for $n \geqq 21$. The 15- and 18-car schedules terminated, claiming that no solution existed. This situation presented quite a problem since an 18-car schedule had already been found by the random-assignment technique.

The solution to this dilemma was to look at the system used by the random assigner. It would start over if it decided that a particular solution seemed stuck. The permutation search did not try to restart, presuming that any working round was as good as another and cannot preclude the finding of later rounds. But this presumption was not true. A dead end can be reached in only the second round, for example, by placing cars *A, D, G* in *Heat 1*, cars *B, E, H* in *Heat 2*, and cars *C, F, I* in *Heat 3*. (The lane assignments are not important.) Of the remaining cars, *J* through *O*, no three can be found that have not already raced one of the others. More explicitly, the program tried to solve the

problem using only a 15! state space. The state space is 15! per round, and in this case we must find schedules for five rounds, so the true state space is $(15!)^5$.

The ability to backup was added to the search heuristic by saving not only the sequence derived for each round but also the state of the permutation-generating algorithm. If the permutation generator reached the final permutation of the series before a sequence was accepted for a round, the heuristic skipped back to the previous round and resumed searching for another valid sequence for that round. Using this revised heuristic, our program quickly found a solution for the 18-car, six-round problem. After running three weeks on an AT&T 3B2/400, it finally found a solution for the 15-car, six-round problem, thus proving the existence of a solution, shown in Table 1. Schedules derived by this program for nine- to 39-car, six-round divisional races have been published in Chapman, Bahill, and Wymore.

UNRESOLVED PROBLEMS

During our 1989 Pinewood Derby, using our manually derived 12-car, six-round schedule, a parent noticed two cars that did not race each other. Subsequent investigations showed that 13 pairs of cars did not race each other. So we ran our program and found a good, but not perfect, solution for this 12-car race. In this solution, all cars race in each lane twice and no car races any other car more than twice; but cars *B* and *I* never race each other. We added a third constraint for nine-car and 12-car races, namely that every car race every other car. Our program searched for six weeks without finding a better solution. We are still looking for a perfect solution for the 12-car race.

Recently, we encountered an interesting phenomenon. When we discussed this problem with our eminent colleagues who did research in scheduling, seven times in a row we heard them say, "Aw, that's an easy problem. I solved that years ago." However, when we asked them to use their algorithms on our 12-car problem, they produced no solutions. When pushed, they said, "Evidently our algorithm is good for our problem, but it cannot stretch to your problem." After discussing this problem for three years, we have concluded that scheduling problems are indeed difficult. Each researcher spends time and money solving a particular problem. At the end of this endeavor, they hope that their technique is general. However, when a problem comes along that their algorithm cannot solve, they must reluctantly admit that their algorithm only works for a particular problem, not for a general class.

EFFICIENCY VS. FLEXIBILITY

We do not believe that efficiency is the most important property of an algorithm: We think that flexibility is more important. We have many computers, so we have no problem with letting a particular computer compute for five or six weeks finding a solution to our problem, as long as it finds a solution. (Multitasking operating systems and resumable programs let us use the computer for other tasks when necessary.) We realize this is not feasible in all situations, but it is practical in many instances such as our Pinewood Derby. We do not want to solve and resolve a problem with more and more efficient algorithms. Rather, we want to get a reasonable solution with a technique that can be reapplied to different problems. Programming time is expensive, but computer time is very cheap. Our algorithm and heuristic let us write programs quickly for the three problems mentioned under related studies and can be easily applied to problems such as the traveling salesman problem and the Chinese Mailman problem.

When we started writing programs for Pinewood Derby schedules, we had difficulty assigning items to processes with constraints. We investigated several methods of solution. Brute force enumeration was infeasible: It took too long. Integer programming could prove the existence or nonexistence of a solution, but it required resources we did not have. We finally wrote an intelligent enumeration heuristic that solved our scheduling problem. In the process, we also developed an efficient and versatile algorithm for generating the permutations of a sequence.

The intelligent-search heuristic can be easily applied to the solution of other scheduling problems by changing the constraint checking procedures. The only requirement needed to achieve significant improvement over full $n!$ or $(n!)^m$ search is that the constraints are such that single positions of conflict are identifiable and resolvable. If the problem is confined to the permutations of a single sequence or can be cast as such, the heuristic search can find solutions quite rapidly. AI

The authors thank Bill Velez for pointing out Kirkman's Schoolgirls Problem and Suvrajeet Sen for showing how to state our problem as an integer programming problem.

SUGGESTED READING

Rouse Ball, W.W., and H.S.M. Coxeter. *Mathematical Recreations & Essays.* New York, N.Y.: Macmillan, 1962, pp. 267-298.

Montgomery, G. "Improving State Space Searches," *AI Expert,* Mar., 1992, pp. 39-43, and "Mastermind: Improving the Search," *AI Expert,* April 1992, pp. 40-47.

Chapman, W.L., A.T. Bahill, and W. Wymore. *Engineering Modeling and Design.* Boca Raton, Fla.: CRC Press Inc., 1992.

A. Terry Bahill is a professor of systems engineering at the University of Arizona at Tucson. He has authored several books, including *Bioengineering: Biomedical, Medical, and Clinical Engineering* and *Keep Your Eye on the Ball: The Science and Folklore of Baseball.*

William J. Karnavas is a Fellow for neurophysiology in the department of neurosurgery at the University of Pittsburgh Medical School.