Parallel Combinatorial Search

BYLINE

Laxmikant V. Kalé and Pritish Jetley
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{kale,pjetley2}@illinois.edu

SYNONYMS

State space search

DEFINITION

Combinatorial search involves the systematic exploration of the space of configurations, or *states*, of a problem domain. A set of *operators* can transform a given state to a series of successor states. The objective of the exploration is to find one, all or optimal goal states satisfying certain desired properties, possibly along with a path from the start state to each goal. Combinatorial search has widespread applications in optimization, logic programming and artificial intelligence.

DISCUSSION

Given an implicitly defined set, combinatorial search involves finding one or more of its members that satisfy specific properties. More formally, it entails the systematic assignment of discrete values from a finite range to each of a collection of variables. Each member of the set represents a configuration or *state* that the basic elements of the problem domain can assume. Therefore, the set is also called a *state space*. Combinatorial optimization problems lend themselves particularly well to expression in this *state space search* paradigm. The basic components of state space search are the *enumeration* of states and their *evaluation* for fitness according to the criteria defined by the problem. Since the state space of these problems can be extremely large, explicit generation of the entire space is often impossible due to memory and time constraints. Therefore, the set of possible states is generated on the fly, by transforming one state into another through the application of a suitable operator. However, in the absence of a mechanism to detect the generation of duplicates, this procedure might begin exploring a path of infinite length, thereby failing to terminate. This is the problem of *infinite regress*, and can be mitigated by one of two techniques. The first relies on the comparison of a generated state with its ancestors, which are stored either in a distributed table or passed from a parent to its newly-created child state. Another approach constrains the search procedure to consider only cost-bounded solutions. This is the case, for instance, in the Iterative Deepening A* algorithm discussed later. In certain situations, the search may be guided

by a *heuristic* measure of distance between states: those considered closer to the goal may be given priority of consideration over those believed to be more distant.

Searching for All Feasible Solutions

A basic combinatorial search problem is one in which all feasible configurations of the search space are desired. An example is the N-Queens problem: find all configurations of N queens on an $N \times N$ chessboard such that no queen *attacks*, i.e. shares the same row, column, or diagonal, with another. In a particular formulation, a state of the problem is represented by a configuration of the chessboard in which each of the first k rows holds a non-attacking queen and the remaining $N - k$ are empty. The enumeration process can be described as a tree search. The root represents a state in which no queens have been placed on the board. The rest of the search tree is defined implicitly as follows: given a state s with k non-attacking queens placed in the first k rows, its child state has the same arrangement of queens as s, and an additional queen placed in the next empty row. Defining the search tree in this fashion is advantageous: the tree can be preemptively pruned at nodes that cannot possibly yield solutions. In the running example, child states are only generated if they do not produce conflicts with previously placed queens.

The search strategy described above is a type of *multistep decision procedure* (MDP). Each step of the algorithm generates a new state s from its parent p by *deciding* an element of the configuration space. At step $k + 1$ of N-Queens, this element is the position of a queen in row $k + 1$ of the $N - k$ remaining rows. An MDP ensures that no duplicate states are generated. In particular, there can be no recurring states along the path from the start to the current state, so that infinite regress cannot occur.

The tree search defined above is useful even when the configurations are defined differently. Consider the problem of finding a knight's tour on the chessboard: starting at a square and using only legal moves, can a knight visit every square on the board exactly once, returning finally to the starting square? This is a special instance of the Hamiltonian Circuit Problem: given a graph, a circular path must be found that visits every vertex exactly once. Here, the vertices correspond to squares on the chessboard. An edge (u, v) connects vertex u to v such that v can be reached in one knight-move from u. To enumerate all such tours, a tree search might define a state as a knight's path of length k originating at the start. Each child extends the path by adding a move to it.

Variable Selection

Given a particular node in the search tree, there is a choice of which branch to select when considering new children. This is called *variable selection*. Continuing with the N-Queens example, at step $k + 1$, *any* of the remaining $N - k$ rows may be chosen as the recipient of the $k + 1$-th queen, not just the $k + 1$-th row on the chessboard. The size of the tree beneath the current node (and therefore the effort involved,) can be significantly affected by the choice of branching variable. A good heuristic is to select the most constrained variable at each step. In the N-Queens example, this would mean placing the next queen in the row with the fewest non-attacked squares.

Parallelization

In a sequential search, the tree is typically explored in its entirety by a depth-first procedure. A stack is used to hold unexplored children at each level. Given a search tree of depth d and branching factor b, this only requires $O(bd)$ memory, in contrast to the $O(b^d)$ size of the search space. A

parallel implementation of this search procedure requires the distribution of work into discrete chunks called *tasks*. To ensure the efficient execution of these tasks on processors, the inter-related issues of task creation, *grainsize* control, and load balance must be considered. Grainsize can be defined roughly as the ratio of computation work to number of messages sent. There is a certain overhead in creating tasks, and a separate overhead if the task description is moved to another processor. Thus, it is important to keep the *average* grainsize above a certain threshold to limit the impact of parallel overhead. At the same time, no single task should be so large as to make all processors wait for its completion.

A parallel search may define a task as the subtree beneath a single tree node. Grainsize estimation then requires a simple metric which is correlated to the computational cost of a node. In N-Queens this could be the number of queens that remain to be placed. It is not an exact measure: sometimes a node with many queens remaining may still generate very little work under it, because of the impossibility of finding a solution there. With a suitable metric formulated, a threshold amount of work may be set, below which new tasks are not created; such subtrees are explored sequentially. Continuing with the N-Queens example, an efficient serial backtracking mechanism may be employed when, say, only $m \leq N$ queens remain. The threshold m could be estimated by the exploration of small parts of the search space.

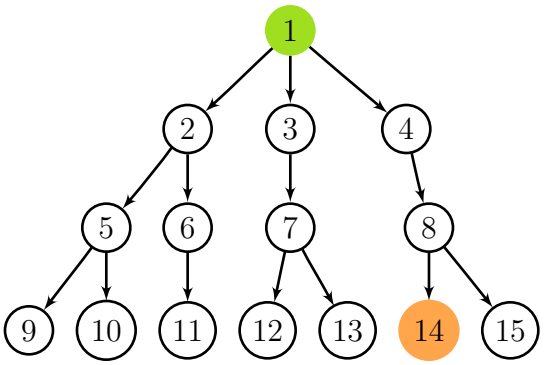
Another technique defines a task as a set of frontier nodes. The exploration proceeds in a depth-first manner, maintaining an explicit stack. At some point, the stack may be split in two (or more) pieces, with each piece assigned to a new task on a possibly different processor. Typically, nodes near the bottom of the stack are used to create a new task. However, *vertical splitting* has also been proposed, wherein half of the unexplored branches at each level of the stack are picked to form starting positions for a new task. This strategy works well for irregular search spaces, but can be expensive for deep stacks.

The literature discusses two methods to determine *when* stacks are split. The first combines grainsize control with the load balancing strategy: in "work-stealing" a processor's stack is split upon receipt of a request for work from an idle processor. The idea was first described by Lin and Kumar [**?**] and later formalized by the Cilk system [**?**]. An alternative, due to Kalé *et al.* [**?**], aims to separate the two issues: the amount of work done by a task is tracked. Once an amount of work above a certain threshold T has been performed, the stack is split into k pieces, each assigned to a new task. This ensures that the grainsize is $T/(k+1)$. However, care must be taken to avoid long chains of large tasks, i.e. when exactly one child has significant work, whereas others do not.

Load Balancing

Early literature classifies load balancers as *sender-* or *receiver-initiated*. Work-stealing is a receiver-initiated load balancing technique. When a processor has no work, it steals work from a randomly chosen processor. It has been shown that random stealing of shallow nodes is asymptotically optimal, in that it leads to near-linear speedups [**?**]. The stealing mechanism could vary from messaging (on distributed memory systems) to exclusive shared queue access. In contrast, sender-initiated schemes assign newly spawned tasks to some processor, either randomly, or based on a load metric such as a queue size.

This taxonomic distinction fails to hold for load balancing schemes where each processor monitors the size of its own queue and that of its "neighbors", periodically balancing them. Initially, a created task is placed on the creating processor's queue. Unlike random assignment, this avoids unnecessary communication. Depending on the thresholds used to exchange queues and the pe-



(a)

t	p_1	p_2	p_3
1	1		
2	2	3	4
3	5	7	8
4	9	12	14

(b)

Figure 1: Acceleration anomaly in parallel depth-first search: **??** illustrates the state space tree of a problem. Search begins at node 1; node 14 is the goal state. A serial depth-first search for the first solution takes 14 steps; **??** shows a schedule for the parallel depth-first search of the same tree with 3 processors. The search takes 4 steps, for a speedup of $14/4 = 3.5 > 3$.

riodicity of load exchanges, such schemes can tune themselves effectively. At one extreme, they can approximate work-stealing, where neighborhoods are global, an empty queue is used as a trigger, and rebalancing is done by selecting a random neighbor. Another advantage is their proactive agility: they increase efficiency by moving work before a processor goes idle. However, these schemes create more communication traffic.

Searching for Any Feasible Solution

Unlike the N-Queens or graph coloring problems where the objective is to identify *all* ways of satisfying given constraints, certain situations require the generation of *any* satisfactory configuration. A classic example is the 3-SAT problem. **Value ordering** becomes an important heuristic in this context: given b children of a node, explore the subtree of that child next whose subtree is most likely to contain a solution. Problem-specific figures of merit are used to order children. For instance, a simple 3-SAT heuristic might rank the two values possible for a variable (true or false) in decreasing order of the number of clauses satisfied by each.

The same parallelization techniques as outlined for the all-solutions search may be used, modifying them to terminate when a solution is found. However, note the *speculative* nature of the methods in this context: regions of the tree that may not be visited in a sequential procedure are searched concurrently in the hope that a solution may be found quicker. This can lead to anomalies in parallel performance [**?**], because nodes are not visited in the same order by the parallel algorithm. In the worst case, an added processor may generate useless work for other processors in the form of nodes that do not yield solutions. This situation is referred to as a *detrimental speedup anomaly*. On the other hand, as illustrated in Figure **??**, this addition of resources could lead to superlinear speedups (the *acceleration anomaly*.) Even on a fixed processor count, variations in the timing of load balancing could lead to widely differing execution spans between runs. Kalé *et al.* [**?**] obtain consistent speedups by prioritizing node exploration in the following manner: if a node with k children has priority q, each child's priority is obtained by appending its $\log k$ bit rank to q. This leads to a lexicographic ordering of tree nodes. Therefore, all descendants of a node's

left child have higher priority than the right child's descendants. A prioritized queue (on a shared memory machine) or a prioritized load balancing scheme is used to steer processors towards high priority work. This scheme also leads to low memory usage: the search frontier forms a characteristic broom shape that sweeps the state space in accordance with the ordering on the nodes. Assuming a constant branching factor, the total memory usage with p processors is $O(p+d)$ rather than the $O(pd)$ requirement of the depth-first tree. On distributed memory machines, achieving this bound depends on the quality of the prioritized load balancer. Furthermore, since such schemes move work from the left part of the tree to all processors, they tend to have a sizable communication overhead. Therefore, a simple depth-first search may be more suitable in some cases.

Searching for an Optimal Solution

Some problems assign measures of fitness to feasible solutions. This renders certain solutions "better" than all others by some metric, so that the objective becomes the search for an *optimal* solution. Examples include integer programming, solving the 15-puzzle (or Rubik's cube) with the fewest moves, and the search of a graph for a least cost Hamiltonian cycle. One could use the all-solutions methods discussed previously, and then select the best solution among all found. However, this is typically wasteful, and more efficient search methods are available.

A* Search and Iterative Deepening

In many problems, it is possible to define a heuristic function that, given a node n, computes a lower bound on the cost of any solution in the subtree beneath it. Such a function is called an *admissible* heuristic. For instance, in the 15-puzzle, the number of tiles that are out of place is an admissible heuristic: at least that many moves are needed to attain the goal state, since only one tile is shifted per move. (The "Manhattan distance" of each tile from its final position is a stronger heuristic.) The lower bound on the *total* cost of a node is the sum of this value and the cost of arriving at it from the start. If unopened nodes are processed in ascending order of their lower-bounds, it can be shown that the first solution found is optimal [**?**]. This is called the A* search procedure. The A* strategy leads to an exponentially sized node queue; by contrast, depth-first search is highly memory efficient. The Iterative Deepening A* (IDA*) technique developed by Korf [**?**] combines the best properties of the two. It is applicable when the solution cost is quantized. For example, the number of moves needed in the 15-puzzle is an integer. If there is no solution of d moves, the procedure looks for a solution of $d+2$ moves. (Because of parity arguments, the number of moves needed for a given starting state is known to be either even or odd.) A depth-first search bounded by a cost d may then be organized. An admissible heuristic is used to stop the search below nodes with cost lower bounds greater than d. If there is no solution of cost d, the bound is increased to the next possible value ($d+2$ for the 15-puzzle), and the search is restarted. Although there is duplication of the higher levels of the search tree, this technique is asymptotically optimal in the amount of work done [**?**]. Further, it offers control over the amount of memory used, while maintaining the property that the first solution found is the optimal one. Moreover, this method precludes the problem of infinite regress without the need for duplicate detection.

An effective parallelization of IDA* is described by Kalé *et al.* [**?**]. It uses bitvector prioritization to overlap execution of multiple iterations with different bounds. This ensures that the last iteration (the one in which the solution is found) is executed in a way that minimizes speculative

loss, leading, as before, to consistent and monotonic speedups. In addition, the latter part of one iteration, where it winds down causing low processor utilization, is speculatively overlapped with the start of the next, thus improving efficiency.

Branch-and-Bound

Optimization problems can also be solved using the branch-and-bound technique, wherein properties of partial solutions are used to discard infeasible portions of the search tree. This can reduce the effort expended in finding optimal solutions. The main components of this mechanism are the *branching* and *bounding* procedures. Given a node n in the search tree, the branch procedure generates a finite number of children. The bounding procedure assigns to each child a cost bound that determines when the child is explored, and whether it is explored at all. Nodes are pruned if they need not be explored. In addition, the bounding function must be *monotonic*: the bound of a node may be no better than the bound of its parent. An *exploration* mechanism is required to choose between the children of a node generated at each step. Such a mechanism may prioritize children based on their depth or their estimated ability to yield a solution. The procedure terminates when all subproblems have either been explored or pruned.

Consider the Traveling Salesman Problem. Whereas more efficient bounding techniques have been discussed in the literature, the following naïve procedure is presented for the purpose of exposition. A node n represents a partial solution comprising paths between cities such that they form a (possibly incomplete) tour. The children of n are enumerated by listing cities which can be visited from the most recently added destination. A partial solution has a cost bound equal to the length of the path that it represents. This is a monotonic lower bound, since the cost of a path through a child of n is at least as great as the cost of a path through n itself. The algorithm tracks the cost c of the cheapest complete tour encountered up to a certain point in the search. Notice that the tree can be pruned at children with lower bounds *greater* than c. Starting with an empty tour, all possible tours may be considered using this procedure.

The basic data structure in the branch-and-bound is a prioritized queue that stores the nodes comprising the frontier of the search. Anomalous speedups can result if these nodes are processed in an unordered fashion. The literature suggests *unambiguous* heuristic functions [**?**] for shared queues of nodes. Such a function h differentiates between nodes based on their values, so that for nodes n_1 and n_2, $h(n_1) = h(n_2) \Leftrightarrow n_1 = n_2$. Even with consistent speedups, performance is bound by queue locking and access overheads. These can be mitigated by the use of concurrent priority queues.

In distributed memory implementations, the frontier may either be stored in a centralized or distributed manner. The former strategy usually engenders *master-slave* parallelism, where the master processor distributes work from a central node pool to worker processors. This approach retains complete information about the global state of the search, thereby enabling an optimal exploration of the frontier without any speculative work. However, it is not scalable due to the bottleneck at the master. Distributing the node pool affords more autonomy to individual processors. Efficiency can be increased if a processor broadcasts newly encountered lower bounds to others. This helps other processors discard nodes that cannot yield optimal solutions. However, since it is hard to track node quality, effort might be wasted in exploring infeasible nodes for want of accurate bound information. *Quality equalization* may be performed to reduce speedup anomalies and distribute useful work equitably. This is either done periodically or when an associated trigger is activated.

Equalization involves the movement of promising nodes between processors, which can be done in a hierarchical fashion to reduce communication. Grama and Kumar [**?**] and Kalé *et al.* [**?**] survey such parallel branch-and-bound techniques.

Bidirectional Search

Problems such as Rubik's Cube stipulate *a priori* the exact configuration of the goal state. In such problems, a *bidirectional* search may be employed to construct a path to the goal from the start state. By initiating two paths of search, one moving forward from the start state and the other backward from the goal, the size of the search space explored can be reduced substantially. On average, a unidirectional search of a tree of depth d and branching factor b visits $O(b^d)$ states to find an optimal solution. In contrast, by starting two opposing searches that meet, on the average, at depth $d/2$, only $O(b^{d/2})$ states are explored.

There are two main ways of organizing the forward and backward searches. The first method uses a backward depth-first procedure to exhaustively explore the tree starting from the goal, up to a certain height h above it. The states generated by this backward search are stored in the *intermediate goal layer*, the size of which is limited by the amount of memory available. The forward search is a (memory efficient) depth-first procedure or a best-first search with iterative deepening. Instead of checking for goal states, the forward search looks for each enumerated state in the intermediate layer. A match indicates the presence of a solution. For distributed memory systems, the intermediate layer may either be replicated on every processor, or distributed across all available processors. Using replicas of the intermediate layer lowers the communication cost of the algorithm, but forces a reduction in the depth of the backward search. Kalé *et al.* [**?**] describe the use of distributed tables in multiprocessor bidirectional search.

A second class of bidirectional search schemes uses best-first techniques in either direction. This approach has its pitfalls: The frontiers of the forward and backward search may pass each other, resulting in two non-intersecting paths (in opposite directions) from the start to the goal. Therefore, instead of doing less work, the algorithm will have performed more work than a unidirectional search, yielding poor performance. To overcome this situation, *wave-shaping* algorithms attempt an intersection between the forward and backward search frontiers. Nelson and Toptsis [**?**] survey bidirectional search and provide parallel variants of pioneering uniprocessor techniques. Pohl presented the original non-wave-shaping uniprocessor bidirectional search. De Champeaux and Sint formulated a wave-shaping algorithm which estimates the distance between the advancing frontiers to encourage an intersection. This was improved upon by Politowski and Pohl to reduce the computational complexity of the heuristic.

Kaindl and Kainz [**?**] provide empirical evidence suggesting that bidirectional search is inefficient not because of non-intersecting frontiers, but because of the effort expended in trying to establish the optimality of the various paths constructed upon their meeting. Even with the most efficient implementations, bidirectional search can sometimes fail to provide the expected speedups, because of the structure of the problem's state space. Consider the game of peg-solitaire. The initial states of the game have many pegs but few spaces to allow jumps, so that there are few available moves. As the balance between free spaces and pegs becomes more even, more moves can be made and the branching factor increases. Towards the terminal stages of the game, few pegs remain on the board and the average branching factor once again reduces dramatically. Therefore, the state space of peg-solitaire does not form a tree that fans out with increasing depth. Conse-

quently, a unidirectional search from the start state visits significantly fewer than $O(b^d)$ nodes, and there is no practical advantage to using bidirectional search.

Game Tree Search

Game-playing can be represented by trees that trace the sequence of moves made by adversaries. Levels of these trees are marked MAX and MIN in an alternating fashion, depicting the moves made by the player and the opponent respectively. The leaves represent the various outcomes of the game, and are assigned values commensurate with their estimated favorability. However, the enumeration of all possible outcomes and paths to their corresponding leaves can be prohibitively expensive. Chess, for example, has on the order of 10^{43} states. Therefore, given a current state, modern game-playing strategies limit the search for good states to a certain *look-ahead* depth d from it. The *minimax principle* posits that under the assumption of rational play, an optimal strategy can be formulated by picking the most favorable child c of the current node n at each turn. A good approximation of the optimal move at n can be computed by evaluating all nodes under it up to a sufficient depth d, and choosing the best child c. Using a depth-first procedure, this requires $O(b^d)$ time and $O(bd)$ space, where b is the branching factor.

Alpha-Beta Pruning

The alpha-beta pruning procedure tracks the values of nodes encountered in the left-to-right, depth-first traversal of the tree to reduce the amount of work done in searching for optimal moves. For this purpose, the procedure tracks the lower bound on projected value of each MAX node (α) and the upper bound on the projected value for each MIN node (β). Consider a MAX node n that has two MIN children l and r. Suppose that the value of its left child l is found to be $v(l) = 15$. Then, the procedure may prune the tree at any children of r once it processes a c such that r is the parent of c and $v(c) < 15$. This pruning reduces the number of nodes significantly. However, the left-to-right order of tree traversal makes it challenging to formulate an efficient parallelization.

Parallel forms of the algorithm address the different kinds of node in the tree. *Principal Variation Splitting* evaluates the leftmost branch of a game tree before allowing the parallel evaluation of sibling nodes. Ideally, the leftmost branch represents the optimal sequence of moves for the player (game tree branches are generally ordered so that the principal variation is the leftmost branch of the tree) and so yields the tightest bounds for the pruning of the rest of the tree, greatly reducing the amount of work done in parallel. Once the leftmost branch l has been scored, sibling nodes are evaluated in parallel using the bounds obtained from l. The siblings of a node n to its right may be sent refinements of bounds as these are calculated by n. The amount of parallelism is limited by the branching factor of the problem. Further, load imbalance between siblings causes synchronization delays.

The *Young Brothers Wait Concept* (YBWC) orders nodes similarly: the first child of a node (the *eldest* brother) must be explored before the others (the *younger* brothers) are examined. This scheme extends parallel evaluation beyond the principal variation nodes. Processors are said to *own* nodes if they are evaluating the subtrees beneath them. Initially, all processors except one, p_0, are idle. Processor p_0 is given ownership of the root node. Idle processors request work from those that have satisfied the YBWC sibling order constraint, i.e. those processors that have evaluated at least one eldest brother n. This establishes a master-slave relationship between the sender and the recipient of work, and marks a *split-point* at N, which is the parent of n. The master and the slave

cooperate to solve the tree under N. Further, a master that has become idle can request work from a slave that has not completed evaluation. Improved bound and cut-off information is shared with collaborating processors. A stronger formulation [**?**] of the parallelism constraint has been used to yield good speedups in chess-playing on distributed memory machines. *Dynamic Tree Splitting* uses a similar collaboration technique in the context of shared memory systems. However, split-points are chosen according to constraints expressed in terms of the α and β values of nodes.

The Deep Blue computer chess system [**?**] used a master-slave approach to parallelism. Upper levels of the tree were evaluated by the master, the slave processors being allotted work as the search grew deeper. To alleviate the performance bottleneck at the master, slave nodes were always kept busy with "on-deck" jobs. Since search was performed using a hybrid software-hardware approach, load was balanced by pushing long hardware searches into software, and by sharing large pieces of work between workers. In 1997, Deep Blue defeated the then-reigning world champion Garry Kasparov.

AND-OR Tree Search

AND-OR trees arise naturally in the execution of logic programs, and in the problem-solving and planning literature within artificial intelligence. The min-max trees used in evaluating two-person games also reduce to AND-OR trees for the special case when the value of a node can only be a win or a loss. Another example of such trees arises in the graph coloring problem, when subgraphs may be colored independently following the coloring of a partitioning layer of vertices.

In logic programming terminology, the top-level query is a conjunction of literals called *goals*. Typically, multiple clauses are available to solve a given literal. Each clause is a conjunction of literals. The evaluation of a query then naturally leads to an AND-OR tree. There are two kinds of node in an AND-OR tree: an AND-node requires solutions to each of its children, whereas an OR-node requires a solution to at least one of its children. Search procedures used for simple state space search may be extended to this case. In particular, to *construct* a solution, an AND-node requires that information be sent up from its subtrees. A number of approaches to the effective organization of this search procedure have been surveyed, chiefly in the context of logic programming, by Gupta *et al.* [**?**].

An interesting issue concerns the dependence between the sub-problems represented by the children of AND nodes. Consider a query such as: $p(a, X), q(b, Y), r(c, X, Y, Z)$. Upper case letters represent variables that are instantiated by a solution. A solution to p may require that X have a value d, and a solution to q may require that Y have a value e. Unless there is a solution to r that also has $X = d$ and $Y = e$, these solutions to p and q are not useful. Therefore, the search space under r is constrained by using solutions produced by p and q. Whereas the subtrees corresponding to p and q can be explored in parallel, for r, a subtree can only be created for each instance produced by the cross-product of solutions to p and q. This "consumer-instance" parallelism is supported by the REDUCE-OR process model [**?**]. Further, to avoid wasted work, the occurrence of duplicate states along distinct paths must be addressed. This issue is separate from the problem of infinite regress described earlier, which arose due to the generation of duplicates along the *same* path. To avoid duplication of effort, the newer state may be made a client for the result generated by the older instance. The issue of duplicates arises in other search patterns as well (i.e. Game Tree Search, IDA*, etc.) In these paradigms, it may be advisable to terminate one of these two paths, depending on their (under)estimated costs.

More Search Techniques

Several search techniques exist in addition to the ones described in this chapter. Path finding in the context of imperfect graph information is done using dynamic algorithms such as D* and LPA*. Little work has been done on the parallelization of these techniques.

Many *metaheuristic* techniques have been developed. The metaheuristic approach uses generalized search mechanisms, usually inspired by physical and natural phenomena, in lieu of problem-specific heuristics and techniques. Examples include Genetic Algorithms, Simulated Annealing, Local and Tabu Searches, etc. The use of graphics processors as offload devices to aid the solution of such state space search problems has also been considered recently.

RELATED ENTRIES

CHARM++
Cilk
Concurrent Prolog
Parlog

BIBLIOGRAPHIC NOTES AND FURTHER READING