International Journal on Artificial Intelligence Tools, Vol. 1 No. 3 (1992) 369-397

COMBINATORIAL SEARCH ALGORITHMS WITH META-CONTROL: MODELING AND IMPLEMENTATIONS

BENJAMIN W. WAH and LON-CHAN CHU

Center for Reliable and High Performance Computing, Coordinated Science Laboratory
University of Illinois at Urbana-Champaign, 1101 West Springfield Avenue, Urbana, Illinois 61801, USA

Received 29 January 1992

ABSTRACT

In this paper, we model search algorithms with meta-control, allowing resource constraints, approximation, and parallel processing to be incorporated easily in the search process. The basic building block of the model is a hierarchical search process (HSP) consisting of context-free and context-sensitive grammars classified according to problem-independent and problem-dependent parts. The context-sensitive components are used mainly for evaluating decision parameters and in ordering production rules in the context-free grammar. The execution of the grammars for given initial conditions may invoke other HSPs already defined in the system. We describe ISE (acronym for *Integrated Search Environment*), a tool that implements hierarchical searches with meta-control. By separating the problem-dependent and problem-independent components in ISE, new search methods based on a combination of existing methods can be developed easily by coding a single master control program. Further, new applications solved by searches can be developed by coding the problem-dependent parts and reusing the problem-independent parts already developed. We describe the organization of ISE and present experiments carried out on the system.

Keywords: Approximation search, heuristic search, hierarchical search process, meta-control, real-time constraints, and resource constraints.

1. Introduction

Search, an important and powerful method for solving problems in artificial intelligence and operations research, has been modeled extensively in previous research. The earliest model for representing problem solving is due to Karp and Held [7], who developed the discrete decision process using finite automata for solving discrete optimization problems. Gnesi, Martelli, and Montanari [5] later formulated search as the traversal of an acyclic AND/OR graph in a bottom-up fashion. Kumar and Kanal [11] developed the *composite decision process* (*CDP*) that models search spaces by context-free grammars. Recently, Helman [6] formulated search as an algebraic combinations of conjuncts.

Previous schemes aim at modeling solution spaces or search spaces but not search algorithms; hence, they are models of search problems from different aspects.

These schemes can be augmented to represent search algorithms, although the resulting representations are usually too complex to be useful, especially for searches involving meta-control. Searches with meta-control include iterative or recursive searches built from other search methods, and searches under resource constraints. Examples of the former include iterative deepening A* (IDA*) [8], dynamic-control IDA* (MIDA*) [18], DFS* [15], and iterative refining A* (IRA*) [14] while examples of the latter are real-time A* (RTA*) [9], time-constrained A* (TCA*) [16], time-constrained guided depth-first search (TCGD) [17], and real-time search (RTS) [3].

Our goal in this paper is to develop a basic module, called the *hierarchical search process* (*HSP*), that can be used as a building block of search algorithms with meta-control. We represent such search algorithms as a hierarchy of invocations of HSPs. This representation is natural because many new and efficient search algorithms can be built by composing existing search algorithms. Instead of representing such composite search as a single task, modeling using HSPs provides a uniform way for representing meta-control of more primitive searches and for providing standard interfaces between calls and returns of primitive searches. Such a representation corresponds to the notion of software reusability, in which complex searches can be built by calling primitive searches in a library.

The HSP we develop can be decomposed according to the type of formal languages that are used to represent its functions. We have chosen to decompose a module into components that can be represented by context-free and context-sensitive grammars. The context-free component represents the framework of the search, while the context-sensitive component evaluates the state of the search and provides guidance to the context-free component. The latter may include the evaluation of parameters for controlling the search, ordering of production rules in the grammars, and evaluating solutions based on parameters of the problem instance solved. All these elements are related to the dynamic state of resources and past performance of the search in progress.

The separation of the grammars for representing functions in a HSP into context-sensitive and context-free components defines clearly information that can be varied according to the environment or the problem instance being solved. Context-sensitive components include information on resources available and properties that depend on parameters of the problem instance being solved (such as the computation of the lower-bound value), whereas context-free components define the search algorithm or problem. This separation, therefore, distinguishes definitions from implementation details.

The context-free and context-sensitive components are further decomposed into a problem-dependent part, like branching and evaluation function, and a problem-independent part, like search control and performance monitoring. This separation simplifies the development of new search algorithms and the support of new applications using search. In developing a new search algorithm, only the problem-independent parts need to be added, and all existing applications can be used to

test the new search method. On the other hand, in developing a new application using search, only the problem-dependent parts need to be enhanced, and all existing search algorithms can be applied and tested.

Our implementation of the HSP is *ISE* (acronym for *Integrated Search Environment*), a versatile tool for applying search on different applications and for developing and testing new search methods [2]. The design goals of ISE include a) partitioning of search algorithms and applications in such a way that new application-independent search methods can be added without affecting existing ones and that new applications can fully utilize search methods already coded, b) collection of statistics for performance evaluation and tuning, and c) provision of a user-friendly interface. Our system allows complex meta-control above the original branch-and-bound algorithm to be modeled, especially those involving dynamic resource control. By compiling routines from the problem-dependent and problem-independent directories, a single program implementing various search algorithms for solving a given application can be generated.

ISE supports finding solutions for optimization and decision problems. An optimization problem entails the search of the optimal solution in the feasible-solution space, whereas a decision problem requires a "yes" or "no" answer to the question raised. The search methods supported by ISE include approximate, resource-constrained, sequential, or concurrent OR-tree and AND-tree searches. By using appropriate primitive searches, AND/OR-graph searches can also be supported.

This paper is organized into five sections. Section 2 describes the background on search problems. Section 3 describes our model and illustrates it by representing various search algorithms with meta-control. Section 4 describes the software architecture of ISE. We present a) issues on search control, memory management, user interface, and concurrent searches, b) methods for coding search algorithms and applications, and c) our experience on programming and experimentation. Finally, concluding remarks are drawn in Sec. 5.

2. Background

In this section, previous work on search is described. The first subsection presents terminologies in search. This is followed by a description of methods for modeling and representing search problems, and techniques for assessing the quality of a search algorithm.

2.1. Search

Search is the process of traversing a search tree or search graph in order to find a solution that optimizes the given objective, while satisfying the given constraints. The branch-and-bound (B&B) search [10] is a general search algorithm consisting of four components: a) selection rules that select the node to expand next; b) pruning rules that prune inferior nodes according to approximation degree, thresholds, or dominance; c) decomposition rules that expand a search node; and d) termination

rules that determine when the search should be stopped. Note that each of these rules may involve problem-independent and problem-dependent elements.

A B&B search can be used as a building block for more complex searches employing meta-control. For example, an IDA* search [8] can be formulated as a sequence of depth-first B&B searches, while the RTS algorithm [3] can be considered as a sequence of guided depth-first B&B searches with proper selection of pruning mechanisms in between two consecutive searches. Meta-control, like the basic components of a B&B search, may involve problem-independent and problem-dependent components.

The problem-independent component of a search can further be divided into context-free and context-sensitive parts. Rules in the context-free part are independent of the environment, while rules in the context-sensitive part depend on the state of the environment and/or progress of the search. Here, the environment refers to the state of the dynamic variables involved in controlling the search, resources remaining, and historical behavior. For example, in the RTS algorithm [3], the selection of pruning mechanism is context-sensitive because it depends on the resource remaining and the past performance observed. In contrast, the meta-control of RTS algorithm that specifies a sequence of guided depth-first searches is context-free.

Similarly, the problem-dependent component can also be divided into context-free and context-sensitive parts. For example, a decomposition rule may involve selecting an attribute of a search node to decompose (which is context sensitive) and break a node into smaller nodes (which is context free). Similarly, a pruning rule may involve context-sensitive parts (such as the incumbent value collected at run time) and context-free parts (which terminate a node by approximation or thresholding).

In general, a search algorithm can be considered as a hierarchy of search processes, each controlled by processes up in the hierarchy. For this reason, we call a search process in this hierarchy a *hierarchical search process* (*HSP*). The above discussion also shows that a HSP can be divided into four components with respect to problem dependency and context sensitivity. The HSP is discussed formally in Sec. 3.

2.2. Model of search problems

The *discrete decision process* (DDP) of Karp and Held [7] is a ternary tuple. DDP = (Σ, Ω, ϕ), where Σ is a finite alphabet, Ω is the set of all feasible strings from alphabet Σ, and ϕ is an assessment function defined over Ω. ϕ can be real-valued or integer-valued for optimization problems, or can give a "yes" or "no" answer for decision problems. In traditional optimization problems, Σ represents the set of *solution entities* that are elements for constructing solutions, Ω corresponds to the space of feasible solutions, and ϕ is a cost or utility function. Note that DDP is a model of problems rather than a search process, since the method for generating Ω is not defined.

To model the generation of Ω, Kumar and Kanal proposed a *composite decision process* (CDP) [11], consisting of a context-free grammar as the key component. A context-free grammar is a 4-ary tuple $G = (N, T, P, S)$, where N is a finite set of nonterminal symbols, T is a finite set of terminal symbols, P is a finite set of production rules, and S is the start symbol. Each production rule is of the form $\alpha \rightarrow \beta$, where α is a symbol in N, and β is a string of symbols from $(N \cup T)$. A CDP is a ternary tuple.

$$\text{CDP} = (G(N, T, P, S), \phi^N, \phi^T) , \tag{2.1}$$

where ϕ^N and ϕ^T are assessment functions associated with nonterminal and terminal symbols respectively. Note that the search process is primarily specified by the *generating grammar* G, and that the terminal symbols in the generating grammar are solution entities.

The search process can be characterized by the derivation tree derived from the generating grammar such that each nonterminal node in the derivation tree corresponds to a nonterminal symbol in the grammar, and that branching from this nonterminal node corresponds to a production rule in which the nonterminal symbol appears on the left-hand side. Likewise, each terminal node corresponds to a terminal symbol. Without ambiguity, nodes and symbols are used interchangeably henceforth. Consider nonterminal node n and its corresponding production rule $p_n \rightarrow a_1 a_2 \ldots a_k$, $\phi^N(n)$ is defined as

$$\phi^N(n) \triangleq f_{p_n}(\phi^N(a_1), \phi^N(a_2), \ldots, \phi^N(a_k)) , \tag{2.2}$$

where f_{p_n} is the cost function associated with production rule p_n.

Note that the previous models can express either explicit enumeration (enumerating solutions one by one) or implicit enumeration (enumerating solutions via a search graph). Explicit enumeration may require much more time than implicit enumeration because the former cannot apply pruning to reduce the search space. The DDP represents explicit enumeration since enumeration is carried out over set Ω, whereas the CDP is more like an implicit enumeration since the feasible space can be derived systematically by the generating grammar. In this sense, the DDP is a model for representing problems, whereas the CDP is for representing search spaces.

2.3. Problem transformations

The target problems we study in this paper are *combinatorial search problems* consisting of a set of objectives to achieve and a set of problems and resource constraints. In this paper, we consider only state-space searches.

Combinatorial search problems can generally be classified as *decision problems*, *optimization problems*, and *feasibility problems*. A decision problem returns a "yes" or "no" answer to the question raised; an optimization problem finds the optimal

solution optimizing the objective; and a feasibility problem finds n feasible solutions regardless of quality. Without loss of generality, we only consider minimization problems in our discussion, as all problems can be transformed into minimization ones using techniques described in this section.

2.3.1. Transforming maximization problems to minimization problems

A maximization problem P^M can be characterized by a set of problem variables V^M, a set of problem constraints C^M, and an objective function $\psi^M(V^M)$. The problem is specified as

$$\begin{aligned} &\max_{V^M} \psi^M(V^M) \\ &\text{such that } C^M \text{ are satisfied} \end{aligned} \tag{2.3}$$

Let ϕ be the assessment function in the search, and ϕ^T, ϕ^N, ϕ^U, and ϕ^L be the corresponding assessment functions for terminals, nonterminals, upper bounds, and lower bounds respectively.

A maximization problem can be transformed into a minimization problem P^m that is characterized by a set of problem variables $V^m = V^M$, a set of problem constraints $C^m = C^M$, and an objective function $\psi^m(V^m) = -\psi^M(V^M)$. A minimization problem is, therefore, specified as

$$\begin{aligned} &\min_{V^m} \psi^m(V^m) = \min_{V^M} -\psi^M(V^M) = \max_{V^M} \psi^M(V^M) \\ &\text{such that } C^m \text{ are satisfied .} \end{aligned} \tag{2.4}$$

An example of an assessment function is the approximation degree, which for minimization problems is formulated as

$$\varepsilon^m = \frac{U - L}{|L|} , \tag{2.5}$$

where U and L are the upper and lower bounds respectively. For a minimization problem, the lower bound is an under-estimate of the solution value, and the upper bound may be either an over-estimate or the value of a feasible solution. In contrast, for a maximization problem, the upper bound is an over-estimate of the solution value, and the lower bound may be either an under-estimate or the value of a feasible solution. To retain the same interpretation of the degree of approximation ε^M for maximization problems, ε^M is formulated as

$$\varepsilon^M = \frac{U - L}{|U|} . \tag{2.6}$$

Besides negating the objective function in transforming a maximization problem to a minimization problem, the upper and lower bounds need to be negated as well. The degree of approximation ε^x for the transformed problem (i.e., a minimization problem) becomes

$$\varepsilon^x = \frac{(-L) - (-U)}{|-U|} = \frac{U - L}{|U|} = \varepsilon^M , \tag{2.7}$$

where $-L$ and $-U$ are, respectively, the upper and lower bounds in the transformed problem.

2.3.2. Transforming decision problems to optimization problems

A decision problem is one whose solution is either "yes" or "no." Once a solution is found, the problem is solved. Define a *completeness function* $F(s)$ to measure the completeness of a proposed solution s such that

$$F(s) = 1, \quad \text{if } s \text{ is a feasible solution ;} \tag{2.8a}$$

$$F(s) = 0, \quad \text{if } s \text{ is the root ;} \tag{2.8.b}$$

$$F(s) = \frac{1 + F(s')}{2}, \quad \text{if } s \text{ is not a feasible solution} \tag{2.8.c}$$

Note that s' is a parent node of s during the search. In this way, solving a decision problem is equivalent to solving a maximization problem with $\phi^T(s)$ defined in Eq. (2.8.a) and $\phi^N(s)$ defined in Eqs. (2.8.b) and (2.8.c). The optimal value is 1.

2.3.3. Transforming feasibility problems to optimization problems

A feasibility problem is one that finds n feasible solutions regardless of their quality. Once these solutions are found, the problem is solved. Define a *completeness function* $F(s)$ to measure the completeness of the n solutions such that

$$F(s) = \frac{k}{n}, \quad \text{if } s \text{ is the } k^{\text{th}} \text{ feasible solution ;} \tag{2.9.a}$$

$$F(s) = 0, \quad \text{if no solution is found yet .} \tag{2.9.b}$$

In this way, solving a feasibility problem is equivalent to solving a maximization problem with $\phi^T(s)$ defined in Eq. (2.9.a) and $\phi^N(s)$ defined in Eq. (2.9.b). The optimal value is 1.

2.4. Methods for assessing solution quality

There are four popular metrics for assessing solution quality: approximation, accuracy, completeness, and belief. In the following, we assume the assessment values are positive.

Degree of Approximation. The *degree of approximation* ε indicates the deviation of the solution s obtained from the optimal solution s^*. For a minimization problem,

$$\varepsilon \triangleq \frac{\phi^T(s) - \phi^T(s^*)}{\phi^T(s^*)} , \tag{2.10}$$

while for a maximization problem,

$$\varepsilon \triangleq \frac{\phi^T(s^*) - \phi^T(s)}{\phi^T(s^*)} . \tag{2.11}$$

In minimization problems, the pruning rule due to approximation determines that symbol s_1 is inferior to symbol s_2 and prunes s_1 when

$$\phi^L(s_1) \geq \frac{\phi^U(s_2)}{1+\varepsilon} , \tag{2.12}$$

where ϕ^L and ϕ^U denote the lower and upper bound functions respectively. Likewise, for a maximization problem, s_1 can be pruned when

$$\phi^U(s_1) \leq \phi^L(s_2)/(1-\varepsilon) . \tag{2.13}$$

The solution obtained is guaranteed to have approximation degree ε if the search was started with an approximation degree ε and used pruning rule (2.12) (or (2.13)) for minimization (or maximization) problems. In fact, the actual solution obtained should have an approximation degree better than or equal to ε because the optimal solution is unknown during the course of the search process, and $\phi^U(s_2)$ in Eq. (2.12) for minimization problems (or $\phi^L(s_2)$ in Eq. (2.13) for maximization problems) is usually inferior to the optimal solution. The degree of approximation of the optimal solution is 0.

Degree of Accuracy. The *degree of accuracy* α indicates how close is s, the solution obtained, to s^*, the optimal solution. For a minimization problem,

$$\alpha \triangleq \frac{\phi^T(s^*)}{\phi^T(s)} , \tag{2.14}$$

while for a maximization problem,

$$\alpha \triangleq \frac{\phi^T(s)}{\phi^T(s^*)} . \tag{2.15}$$

In minimization problems, symbol s_1 is considered inferior to symbol s_2 with respect to the degree of accuracy and can be pruned if

$$\phi^L(s_1) \geq \alpha \times \phi^U(s_2) . \tag{2.16}$$

Likewise, for maximization problems, s_1 is pruned when

$$\phi^U(s_1) \leq \frac{\phi^L(s_2)}{\alpha} . \tag{2.17}$$

The solution obtained is guaranteed to have accuracy degree α if the search was started with an accuracy degree α and used pruning rule (2.16) (or (2.17)) for minimization (or maximization) problems. The solution obtained should have an accuracy degree better than or equal to α because the optimal solution is unknown during the course of search, and $\phi^U(s_2)$ in Eq. (2.16) for minimization problems (or $\phi^L(s_2)$ in Eq. (2.17) for maximization problems) is usually inferior to the optimal

solution. The accuracy degree is a value between 0 and 1 if ϕ^T, ϕ^L, and ϕ^U are non-negative. The degree of accuracy of the optimal solution is 1. Note that the degree of accuracy is related to the degree of approximation by

$$\varepsilon = \frac{1-\alpha}{\alpha} \qquad \text{for minimization ,} \tag{2.18a}$$

$$\varepsilon = 1-\alpha \qquad \text{for maximization .} \tag{2.18b}$$

Degree of Completeness. The *degree of completeness* κ indicates the completeness of the solution s obtained by the search process. The solutions obtained may be incomplete because all resources were expended and the search has to be terminated prematurely. Typical examples can be found in robotic routing against deadlines [13]. The degree of completeness is a value between 0 and 1. A complete solution has a completeness degree of 1.

Degree of Belief. The *belief degree* β indicates heuristically or statistically the belief that solution s found by the search is the one desired. This is useful in searching in a fuzzy environment, and when various heuristics are scheduled against a deadline. The degree of belief is a value between 0 and 1. A totally confident solution has a belief degree of 1.

Thresholding. In a minimization problem with threshold θ, a node with lower bound larger than θ is pruned. The pruning rule used in search with thresholding is

$$\phi^L(s) \geq \theta \ . \tag{2.19}$$

Note that searches using degree of completeness or belief can be transformed into searches with thresholding by maximizing the solution value in terms of the completeness or belief degree.

3. Search algorithms with meta-control

In this section, we propose a model for search algorithms with meta-control and illustrate the model by showing the formulations of familiar search algorithms.

A previous model that represents search spaces is the CDP. CDPs are, however, inadequate in modeling search algorithms, especially those with meta-control. A CDP has one context-free grammar for characterizing context-free search control, including the problem-dependent and problem-independent parts. Hence, in a search with meta-control in which one component may call other basic search algorithms, a single context-free grammar implementing the entire control may result in complex interactions between the context-free and context-sensitive parts.

A better model to represent search algorithms with meta-control, such as IDA*, RTS, MIDA*, and IRA*, is a hierarchical model in which grammars associated with each control mechanism are partitioned. In the next section, we present the hierarchical search process.

3.1. Hierarchical search process

We model the basic building block of a search algorithm as a *Hierarchical Search Process* (*HSP*) that consists of four components:

(i) PI/CF: problem-independent context-free component,
(ii) PI/CS: problem-independent context-sensitive component,
(iii) PD/CF: problem-dependent context-free component, and
(iv) PD/CS: problem-dependent context-sensitive component.

Figure 1 shows the four components and their interactions in terms of flow of information from one component to another. The PI/CF component is the entry point to the search, representing the definition (or framework) of the search algorithm involved. It invokes the PI/CS and PD/CS components. The former uses specific problem-independent control information in the search, such as the status of resources, while the latter represents the definition of the application to be solved. The last component, PD/CS, involves application-specific information, such as assessments and past performance obtained in the application.

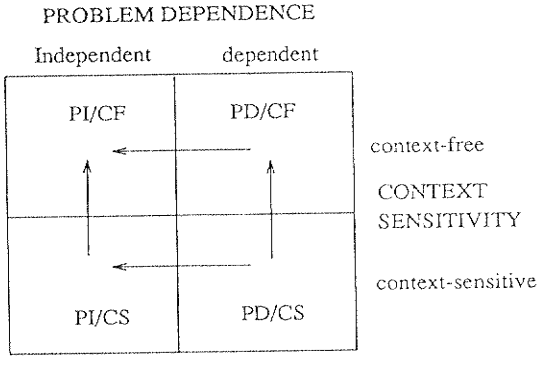


Fig. 1. The four components of a hierarchical search process. Arrows indicate interactions in terms of flow of information from one component to another.

The context-sensitive component in the definition of the HSP can be viewed as a mechanism for selecting production rules specified in the context-free grammars. In other words, the search isolates the context-free grammars from the context-sensitive search controls by employing the latter to carry out context-sensitive resolution on the former. A search can, therefore, be viewed as a tree derived from the generating grammar specifying the PI/CF search control resolved according to the PI/CS component. A leaf of this tree consists of a context-free grammar specifying the PD/CF search control, the PD/CS component, and a finite set of initial conditions.

The execution of the grammars defined in a HSP may result in the invocation of grammars defined in other HSPs. If a HSP does not invoke other HSPs, it is called

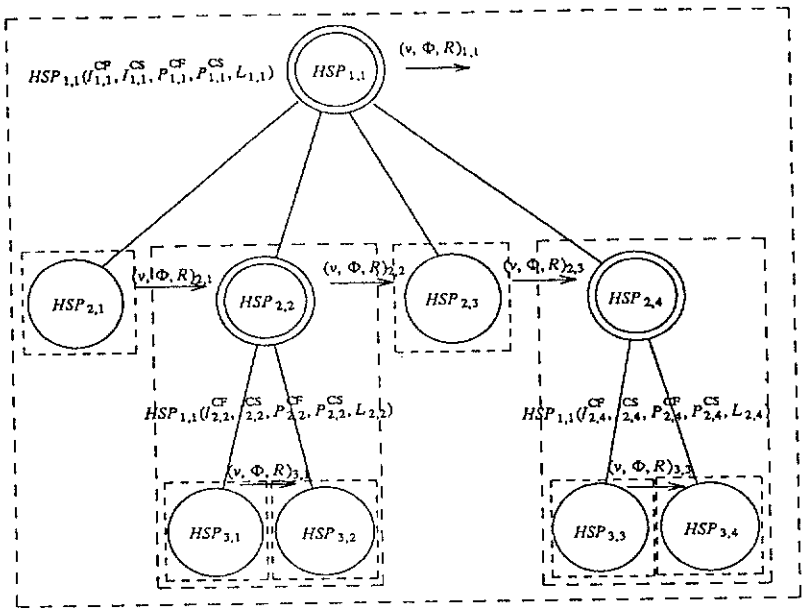


Fig. 2. Hierarchy of calls generated when $HSP_{1,1}$ is invoked. (A dotted box indicates all the HSPs invoked when the root of the subtree is invoked. A horizontal arrow indicates the flow of results from one HSP to another.)

a *primitive search*. Examples of primitive searches are depth-first and best-first searches. Figure 2 shows an example of the sequence of HSPs called when $HSP_{1,1}$ is called. $HSP_{1,1}$ invokes four other HSPs, where $HSP_{2,1}$ and $HSP_{2,3}$ are primitive searches, and $HSP_{2,2}$ and $HSP_{2,4}$ each invokes two other primitive searches.

Formally, a *hierarchical search process* (*HSP*) is defined as a mapping from a quintuplet to a triplet,

$$HSP: \quad (\nu, \Phi, R) \leftarrow HSP(I^{\mathrm{CF}}, I^{\mathrm{CS}}, P^{\mathrm{CF}}, P^{\mathrm{CS}}, L) , \tag{3.1}$$

where I^{CF}, I^{CS}, P^{CF}, and P^{CS} specify, respectively, the PI/CF, PI/CS, PD/CF, and PD/CS grammars, and L is a finite set of initial conditions specifying the environment, such as the initial conditions of the resources and the problem parameters when the search is started. Note that L can be considered as part of the context-sensitive component of the HSP. However, we separate it from the context-sensitive component because we wish to distinguish the context-sensitive component from inputs supplied initially by users.

Each of the context-free grammars in Eq. (3.1) is defined by a 4-ary tuple $(N^{\mathrm{G}}, T^{\mathrm{G}}, P^{\mathrm{G}}, S^{\mathrm{G}})$, where N^{G} is a finite set of nonterminal symbols, T^{G} is a finite set of terminal symbols that are solution entities used in constructing a solution, P^{G} is a finite set of production rules, and S^{G} is the start symbol.

The PI/CF grammar defines the generation of child HSPs. For example, in Fig. 2, the PI/CF grammar of $HSP_{1,1}$ "calls" the PI/CF grammars of $HSP_{2,1}$, $HSP_{2,2}$, $HSP_{2,3}$, and $HSP_{2,4}$.

The context-sensitive components in Eq. (3.1) include PD/CS elements that make decisions based on performance, such as the branching function, and those that schedule the search according to a predefined schedule, such as the depth of the lookahead horizon in the RTA* search. They also include PI/CS elements that schedule the search according to the resources available and past performance, such as the strategy used in MIDA* [18] that sets thresholds based on resources remaining and past performance.

A HSP returns a triplet (ν, Φ, R), where ν is the solution if available, Φ is a finite set of assessment values, such as quality of the solution and any performance estimates, and R is a specification of the resources remaining. The returned triplet can be used by a subsequent HSP as inputs. The arrows in Fig. 2 indicate the flow of results from one HSP to another.

Our formulation defined here is powerful enough to characterize different parsing conditions and search graphs in a top-down or bottom-up fashion. In a top-down search, the original problem representation (symbol S) is decomposed into subproblem representations, which are repeatedly decomposed until a solution is found or infeasibility is proved. It is used in searches like A*, AO*, and branch-and-bound search. In contrast, in a bottom-up search, the solution associated with each symbol is successively refined until the solution associated with the start symbol S is solved. The search can also be characterized by the type of search graph it generates, namely, an OR tree, an AND tree, or an AND/OR graph. This characterization is implicitly represented in the generating grammar.

3.2. Search algorithms modeled using HSPs

In this section, we model a number of familiar search algorithms using HSPs. These include a simple search (depth-first search), an iterative search (IDA*) [8], a resource-constrained search (RTA*) [9], and a resource-constrained iterative search (RTS) [3]. In each case, we model the primitive search and its meta-control as HSPs and describe the PI/CF, PI/CS, PD/CF, and PD/CS components and ternary tuple returned in each HSP. We use the traveling salesperson problem (TSP) as the target application.

The depth-first search is an example of a simple search. Figure 3 shows a depth-first search modeled using HSP in solving a TSP instance. In this example, we have initial conditions that include the pruning mechanism, solution requirements, resource constraints, and inputs. The depth-first search is divided into four components according to the problem dependence and context sensitivity. The returned triplet is not shown in the example.

We use the IDA* as an example of iterative search. In IDA*, the PI/CF component of IDA*'s meta-control HSP consists of a grammar that defines a sequence of

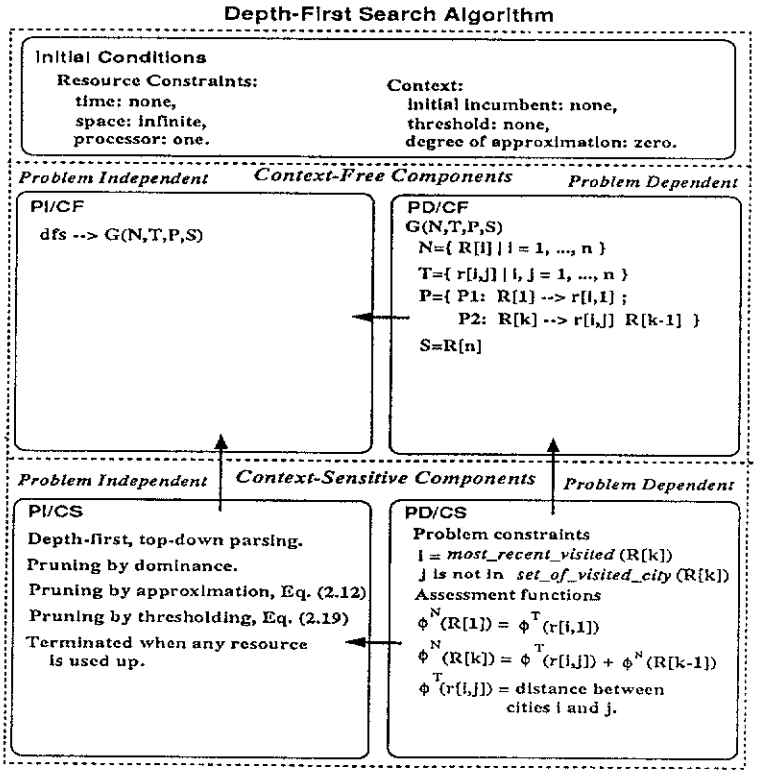


Fig. 3. Example showing HSP of depth-first search in solving an n-city symmetric TSP instance.

depth-first searches. In Fig. 4, we show the HSP used to model the meta-control of IDA*. The pruning mechanism selects a threshold based on a value in the previous depth-first search such that the threshold is equal to the minimum of the lower bounds of children of nodes pruned by thresholding in the previous iteration. The primitive search, or depth-first search, is similar to that shown in Fig. 3, except for the initial conditions, which are shown in Fig. 4.

We use the RTA* as an example to illustrate a time-constrained search. The RTA* itself works like a depth-first search, but at each search node, it looks ahead several levels deeper to obtain a better heuristic estimate for guiding the search. Therefore, during the depth-first search, each node is associated with another secondary depth-first search that corresponds to lookahead. The specification of the secondary search is the same as that in the primary search except for one more pruning mechanism that prunes nodes by depth. This extra pruning mechanism defines the horizon of lookahead.

Figure 5 shows the HSP for modeling RTA* in solving an n-city TSP. Only the initial conditions of the RTA* search and the lookahead search are shown because

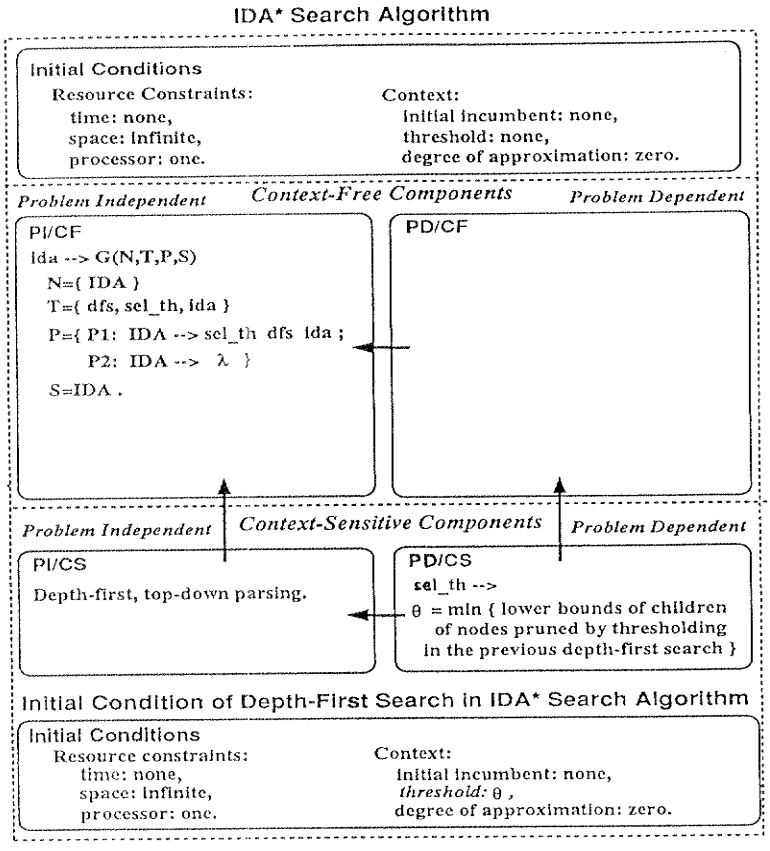


Fig. 4. Example showing HSP of meta-control in IDA* for solving an n-city symmetric TSP instance.

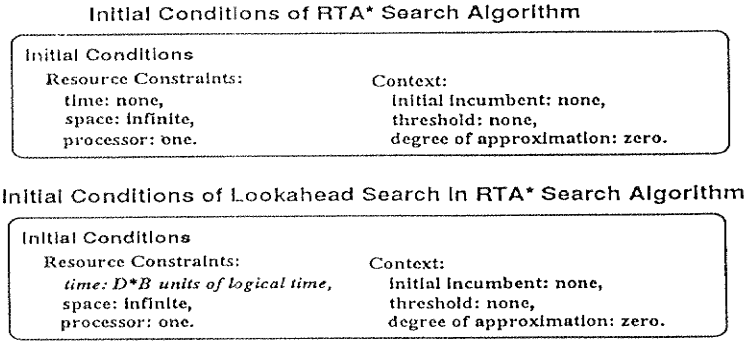


Fig. 5. Example showing initial conditions of HSPs of RTA* for solving symmetric TSP.

the rest is the the same as the depth-first search shown in Fig. 3. Two search parameters, D and B, are used in the initial conditions of the lookahead search. D defines the depth of the lookahead horizon, and B is the average branching degree. The product of D and B defines the amount of time to be expended in the lookahead search.

Finally, we use the RTS as an example to illustrate a resource-constrained iterative search. RTS [3], a search designed to be used in real-time applications, is a generalization of IDA* using approximation and/or thresholding. It is an iterative search that dynamically selects approximation degrees and/or thresholds based on past performance and resources remaining.

The following example shows the HSP model of using RTS to solve a TSP instance against a deadline. In this example, only the time constraint is considered because RTS uses bounded amount of memory. As in the formulation of IDA*, RTS is a sequence of guided depth-first searches and hence, its PI/CF grammar in the meta-control defines this sequence. A noted difference with other searches is that the selection of pruning mechanisms is problem-dependent and context-sensitive because the solution depends not only on resources remaining but also on past performance observed. In Fig. 6, we show the initial conditions of the guided depth-first search; the rest of the guided depth-first search is similar to that shown in Fig. 3 except for the selection function.

4. ISE – An Integrated Search Environment

In this section, we discuss our prototype that implements HSP. Our prototype is developed with two classes of users in mind: a) application-oriented users who wish to either develop new applications using search methods provided or solve problems using searches; and b) search-oriented users who want to develop and test new search methods.

Our objective is to support users so that new applications can be coded easily and that new search methods can be tested with little effort. To accomplish this objective, our design goals of ISE include a) succinct partitioning of search algorithms and applications in such a way that new application-independent search methods can be added without affecting existing ones and that new applications can fully utilize search methods already coded, b) flexible collection of statistics for performance evaluation and tuning, and c) provision of a user-friendly interface.

Our system consists of problem-independent and problem-dependent parts. The problem-independent part includes modules for primitives searches, meta-searches calling primitive searches, and performance monitors. The problem-dependent part implements application-specific functions and procedures called by the problem-independent modules. By compiling and linking appropriate problem-independent and problem-dependent modules, an application-specific object code can be generated. This object code is specific to the application but contains all the different search algorithms implemented. The alternative that compiles all the applications

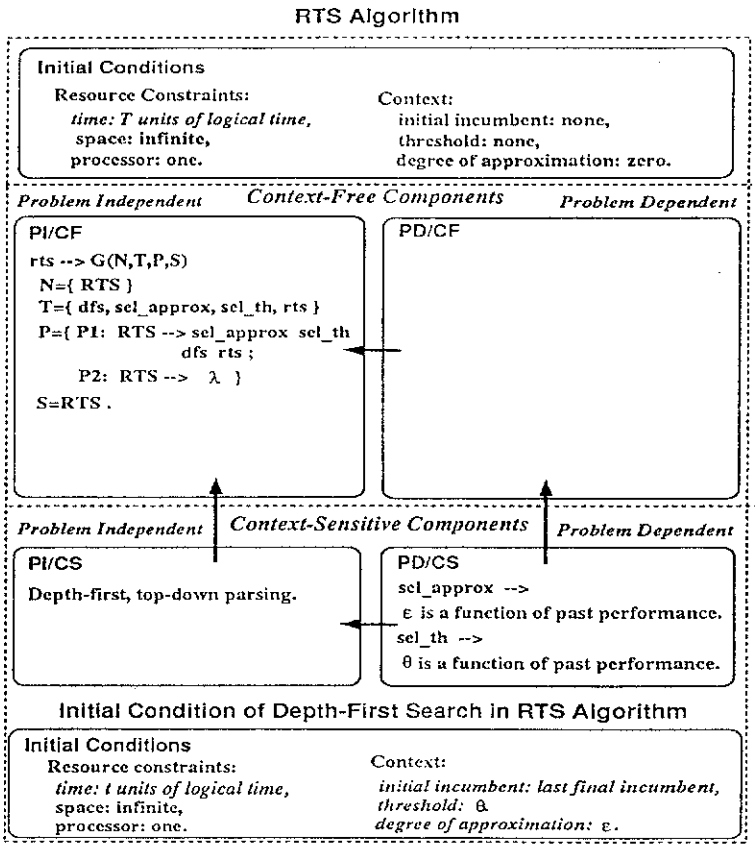


Fig. 6. Example showing HSP of meta-control in RTS for solving symmetric TSP.

into a single program and selecting the appropriate search routine at run time is not chosen because the capability to switch dynamically to different applications is not needed.

For performance tuning of searches and for finding a suitable search method for a given application, our system provides routines for monitoring performance. Performance measure can be either logical or physical. In a search, logical time refers to the number of nodes expanded or generated, while physical time is the processing time expended. Likewise, logical space is the maximum number of active search nodes at any time during the search, while physical space is the maximum memory space used for keeping the active search nodes. Note that logical metrics are machine independent, whereas physical metrics are machine dependent.

Our current system implements only top-down searches. Extensions to bottom-up and bidirectional searches are straightforward, since bottom-up and top-down searches are equivalent, and bidirectional search can be built using top-down and bottom-up searches.

4.1. Software architecture of ISE

The software architecture of ISE, shown in Fig. 7, is the implementation of the logical HSP model. ISE includes two major groups: one related to search control and the other related to peripheral services. Both are represented in terms of the four components according to problem dependency and context sensitivity, each located in one or more physical directories. However, some context-free and context-sensitive components may share the same directory, such as in directories **algorithm**, **primitive**, and **solver**. The reason is that although the component is context free, some parameters used, such as pruning mechanism and resource constraints, are determined from the current context. For efficiency reasons, some of these components are combined into a single routine to avoid excessive procedure calls.

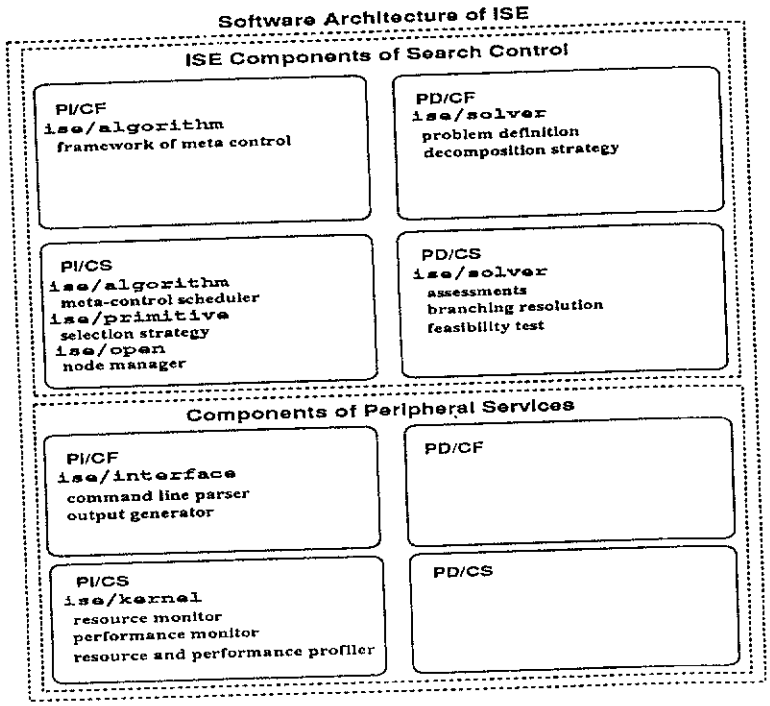


Fig. 7. Software architecture of ISE.

In the rest of this section, we describe the functions implemented in a) search control, which includes codes of search algorithms, primitive searches, and applications, b) resource monitoring and performance profiling, c) memory management, and d) user interfaces for specifying input, profiling, and output commands.

4.1.1. Search control

The search-control group consists of four subdirectories: a) `algorithm` with all search algorithms available in ISE, b) `primitive` with all primitive searches available in ISE, c) `open` containing memory managers for different search algorithms, and d) `solver` including all applications implemented in ISE.

In the PI/CF component, the meta-control of search algorithms are coded. For example, the meta-control of IDA* calls a sequence of depth-first searches, while that of RTS calls a sequence of guided depth-first searches.

The PI/CS component serves three functions. First, it implements the *meta-control scheduler* that selects pruning mechanisms and assigns resources in between two primitive searches based on the past performance and/or resource constraints. Second, it implements the selection strategy that selects a node to expand during the search. In general, a selection strategy can be formulated as a function such that the node with the best functional value is selected. A selection strategy is static if the function only needs to be evaluated once for a node; examples include depth-first and best-first searches. A selection strategy is dynamic if the function needs to be re-evaluated for all active nodes every time selection is done. Third, it implements the node manager that allocates and de-allocates space for each selection strategy. Issues on memory management are discussed later.

The PD/CF component defines the target application and specifies the decomposition strategy. The latter is context free because it does not depend on the state of the search; instead, the expansion is defined by the branching resolution (a PD/CS element) and is related only to the current node to be expanded.

The PD/CS component contains modules for assessment, branching resolution, and feasibility test. Assessment refers to functions for evaluating the quality of a solution or search node. Typical examples include the lower-bound and upper-bound functions. Branching resolution serves as a preprocessor for decomposition and selects an appropriate attribute for decomposition based on the state of the search node.

Currently, the search algorithms implemented in ISE include depth-first search, best-first search, guided depth-first search, generic branch-and-bound search, band search [4], Lawler and Wood's time-constrained approximation search [12], IDA* [8], RTS [3], MIDA* [18], TCA* [16], TCGD [17], DFS* [15], IRA* [14], and some of their parallel versions. The applications implemented in ISE include symmetric traveling salesperson problem (STSP), asymmetric traveling salesperson problem (ATSP), knapsack problem (KS), production planning problem (PP), resource constrained scheduling problem (RCS), vertex cover problem (VC), scheduling for minimizing weighted completion time (WCT), maze solver[a] (MAZE), and N-puzzle solver (PUZZ). These applications have been used in experimenting with the various search algorithms. Note that the problem solvers we implemented are optimization

[a]The maze generator was written based on the maze program in X11R4.

solvers. Decision problems and feasibility problems can be transformed into and solved as optimization problems (see Sec. 2.3).

Table 1 shows a classification of minimization problems with respect to the difficulty in finding feasible solutions as upper bounds. Finding feasible solutions in the applications we implemented in ISE range from easy to impossible. Finding feasible solutions is considered easy when the corresponding algorithm has polynomial complexity (as a function of problem size). Greedy algorithm is a typical method in this case. Finding feasible solutions is hard when the corresponding algorithm requires exponential complexity in the worst case. The exact complexity, however, depends on the constrains posed. If the constraints are tight, then there may be few or no solution in the entire search space, and finding a feasible solution is almost as hard as solving the original problem. If the constraints are loose, then feasible solutions are plenty and can be found in polynomial time. In the third class of applications, there are no constraints imposed, and finding a feasible solution is as hard as solving the original problem. Note that we do not consider problems whose upper bounds can be found by solving a restricted problem. These upper bounds are not useful because they cannot be used as feasible solutions when the search is terminated prematurely.

Table 1. Classification of applications with respect to difficulty in finding feasible solutions as upper bounds.

Class	Class I	Class II	Class III
Degree of Difficulty	Easy (Polynomial Complexity)	Hard (Exponential Complexity or Above)	Not Possible Until Problem is Solved
Examples	STSP, KS, PP, RCS WCT, and VC.	ATSP and MAZE.	PUZZ.

4.1.2. Resource and performance monitoring

These are part of peripheral services (see Fig. 7) designed for resource monitoring and performance profiling. They are application independent but context-sensitive and can be employed for all applications. All the programs are located in the **kernel** subdirectory.

The resource monitor keeps track of resource usage, such as time and space used, cumulated space-time product, and processor utilization, in both physical and logical units. An option is available to specify only a subset to be monitored. The monitor periodically checks whether any resource constraint is violated, signaling the corresponding module when resources are expended.

The performance monitor keeps track of the progress of search, reporting on measures such as the solution quality and the resources used. By specifying an option in the command line, measurements in either physical or logical units can be

collected. The marginal improvement in the quality of solution against the marginal amount of resources expended can be used to trade between performance and cost.

The resource and performance profiler reports performance statistics versus resource usage during the course of the search. In particular, it provides the time used versus various measures, including space used, cumulative space-time product, degree of approximation, incumbents obtained, and maximum space used. The distribution of search nodes by their lower or upper bounds can also be collected. The profiles are generated in such a form that can be used directly by plotting tools.

4.1.3. Memory management

Memory management includes routines for allocating and de–allocating space as well as access to and maintenance of the active list of search nodes.

ISE has a *search-node manager* that maintains a pool of free search nodes in the form of a stack. Whenever space for a free node is needed in the search, the manager is called to allocate space from the pool. Likewise, whenever space for a search node is to be de–allocated, the manager is called to insert the search node into the top of the stack. This scheme favors the "least recently used" replacement strategy in virtual memory and tries to preserve spatial and temporal localities during the search. When the manager runs out of space, it calls the operating system to allocate a new block of space. The size of this block is problem-dependent since the size of a search node may vary over several orders of magnitude. Usually, the block size varies between 256 and 1024 search nodes.

The management of active nodes in a search is algorithm dependent because different algorithms may require different amount of memory space and memory management methods. Search algorithms like best-first search generate a large number of active nodes. In this case, a B+ tree is used because the location and retrieval of the best search node can be done in logarithmic time. Moreover, search nodes with lower-bound values exceeding a threshold can be pruned easily. Other search algorithms like depth-first search use a relatively small amount of space. In this case, the active list is simply a stack, because the most recently generated node will be expanded first and the stack size is bounded by the depth of the search tree. For a guided depth-first search, a linked list is more efficient because it expands a search node, sorts its children by a given criterion, and picks the child with the best value to expand. Here, the number of search nodes in the active list is bounded by the product of the maximum branching degree and the maximum depth of the search tree. In the general case where pruning and selection are carried out using different functions or criteria, it may be more efficient to maintain the active list as a linked list. Maintaining a single $B+$ tree ordered by the selection criterion may result in excessive overhead when pruning is applied to all active nodes. For this reason, some implementations of the B&B algorithm organize the active nodes by the selection criterion and delay pruning until the node is actually selected. More complex management routines, such as those in MA*[1], can also be implemented.

4.1.4. User interfaces

User interfaces are the PI/CF component of peripheral services (see Fig. 7). They are designed to be flexible to be used by programmers and users. There are three levels of interfacing. a) The algorithm-level interface includes protocols and parameter-passing interfaces between the search algorithms and the numerous ISE modules, such as application programs, search primitives, performance/resources profiler, and output reporter. b) The application-level interface includes protocols and parameter-passing interfaces between the application programs and the numerous ISE modules. c) The user-level interface is primarily for the command-line processor and the versatile report generator. All these routines are stored in the subdirectory *interface*.

The command-line processor parses the command line and transforms the command into an internal format of search controls and parameters. Users can define the search algorithm to be applied, algorithm-specific parameters, inputs for specifying one or a set of problem instances, the unit used in measurement (whether logical or physical), random seed used for generating a random problem instance, resource constraints, the quality of the target solution, profiling controls, and output formats. Most parameters have their own defaults and do not need to be specified fully. They can also be specified in any order.

A line-oriented command-line parser is designed in ISE. The command line can be parsed by a regular grammar, called the command-line grammar, whose rules are summarized in Fig. 8. The alternative of using a screen-oriented command-line processor is not taken in ISE because the set of parameters may be problem dependent, and it is difficult to invoke scripts in testing a sequence of searches.

The parameters following the keyword "`:prob`" must be specified in order for ISE to know the problem size, the number of problem instances to solve, and the surrogate, which is a seed used to reset the random number generator.

ISE currently provides five levels of debugging information, increasing in the amount of output as the level is increased. The user can specify the level desired by an integer following the keyword "`:dbg`."In the lowest level, only calls to search algorithms and the solutions returned are reported. In the highest level, debugging information down to the level of branch decisions and bounds evaluated are output.

Users can select one of the following types of outputs after the keyword "`:io`": a) custom-designed output, b) explanatory summary, c) tabular listing, and d) profiling output. The custom-designed output can be any format specified by a user; an example would be one in which the outputs can be piped to another program. The explanatory summary is meant for presenting the results to users. The tabular listing is used by plotting packages. The parameters specified for the output formats can be stored in a file to reduce the complexity in the command line.

Profiling can be collected in either *intra* or *inter* mode. In intra-search-process profiling mode, profiling statistics is collected for a single primitive search algorithm,

```
COMMAND ::= SOLVER PROBLEM OPTION .
PROBLEM ::= ":prob" num_of_instances problem_size surrogate .
OPTION ::= MEASURE | REPORT | SEARCH | CONSTRAINT | QUALITY |
           DEBUG | IO | PROFILE | PARAMETER |λ .
MEASURE ::= ":virtual" | ":real" .
REPORT ::= ":report" report_filename .
SEARCH ::= ":search" search_primitive search_algorithm .
CONSTRAINT ::= ":constr" time [space [cost [ ... ] ] ] .
QUALITY ::= ":approx" quality_degree .
DEBUG ::= ":dbg" level_of_details .
IO ::= ":io" LIST_OF_FILE_NAMES .
PROFILE ::= ":pf" LIST_OF_FILE_NAMES .
PARAMETER ::= ":param" LIST_OF_VALUES .
```

Fig. 8. Command-line grammar of ISE ("[]" means optional parameters, and " | " means alternative ones).

while in inter-search-process profiling mode, profiling statistics is collected over a sequence of primitive searches. The latter mode is important in showing the performance of search algorithms with meta-control, such as RTS and IDA*, which call a sequence of primitive searches, each with its own intra-search-process profile.

4.1.5. Other considerations

In this section, we discuss generalizations to HSPs we have developed in ISE. These include concurrent implementations, and searches based on AND-trees and AND/OR graphs.

Conceptually, the lowermost level of control in a search is at the search nodes. This bottommost level in the control hierarchy corresponds to the *search-node process* (SNP) that handles a single search node during the search. It corresponds to a production rule in the context-free grammar of HSP. In a state-space search, the SNP takes care of decomposing (or branching of) a search node, testing its feasibility, evaluating the bounding functions, pruning, and updating search attributes, such as the incumbent, if necessary. In this way, a search is a sequence of homogeneous or heterogeneous SNPs.

The SNP level is important for concurrent searches, such as parallel searches and bidirectional searches, where multiple search processes for solving the same problem instance can coexist in a cooperative or independent fashion. As a result, it is necessary to distinguish the scheduling at the SNP level from that at the search level, and programmers must implement the scheduling of SNPs explicitly.

For nonconcurrent searches, the SNP is always carried out in a search process and hence, the SNP can be integrated into the search process to avoid the extra overhead of procedure calls.

Readers may notice that all the problems we implemented so far in ISE are solved by OR-tree searches only. Other types of searches, such as AND-tree and AND/OR-graph searches, are not implemented. The reason for this bias is that the new search algorithms we developed [3, 4, 14, 16, 17] handle OR-tree searches only. However, as shown in the following discussion, ISE is not limited to OR-tree searches and can support AND-tree and AND/OR-graph searches as well.

The only difference between AND-tree and OR-tree searches is at the SNP level. To complete an SNP, an AND-tree search requires *all* children spawned by the SNP to be explored in order to obtain a solution, while an OR-tree search only requires *one* child spawned to be explored. To generalize the OR-tree searches we have implemented in ISE, SNPs can be classified into one of two types: AND SNPs and OR SNPs. An AND SNP forms a building block of AND-tree searches, while an OR SNP is for OR-tree searches. The difference of AND-tree and OR-tree searches lies in the type of SNPs they employ.

An AND/OR-graph search is a combination of OR-tree and AND-tree searches with possible overlapping subtrees. In a *canonical* AND/OR-graph, each level consists of either all AND nodes or all OR nodes, and two consecutive levels are of different types. It is known that any AND/OR-graph can always be transformed into a *canonical* AND/OR-graph, and for each node, its AND/OR-subgraph can be transformed into a functionally equivalent AND/OR-subtree with possible redundancies in nodes searched. By checking redundancies of search nodes at the search level, an AND/OR-graph search is conceptually equivalent to an AND-tree search or an OR-tree search.

4.2. Programming and experimenting with ISE

In this section, we describe procedures for programming ISE with respect to implementing new search algorithms and new applications. We also describe how to carry out experiments on ISE.

4.2.1. Implementing a new search algorithm

A search algorithm can be represented by a problem-independent context-free grammar, as shown in the problem-independent component in Fig. 7. The PI/CF component contains the framework of the search algorithm, and the PI/CS component contains the meta-control scheduler and selection strategy.

Before a search algorithm can be coded, it usually requires the following context-sensitive information defined. a) *Resource Constraints.* A combination of time, memory space, processors, cumulative space-time product, and cumulative processor-time product may be specified as constraints. The search is terminated whenever any of these constraints is violated. b) *Pruning Mechanisms.* Degree of approximation, threshold, belief, completeness, and dominance may be specified as pruning mechanisms. c) *Resource/Performance Measures.* Resource and performance measures (in terms of physical or logical units) used at run time to control

the search must be identified. d) *Resource/Performance Profiles.* Run-time profiles, in terms of quality of solutions versus time consumed, and efficiency of using one resource versus another, must be specified.

A search algorithm may use problem-dependent information defined in the applications. This is done by calling problem-dependent routines, which may include the following: a) *decomposition* for decomposing a search node into child nodes, b) *bounding function* for evaluating the application-dependent lower and upper bounds of search nodes, c) *feasibility test* for checking whether a search node is a solution or not, and d) *dominance test* for pruning node y with respect to node x when the best solution in the subtree rooted at node x is better than that in the subtree rooted at node y.

The procedure for programming a new search algorithm can be summarized as follows. a) Model the search algorithm by HSPs, each with four components. There may be several HSPs for a single search algorithm. b) For each component, check whether the necessary routines have already been coded in ISE, and code modules that are not available in ISE. All the compilations for generating the object code for an application are issued by a makefile, as exemplified in Fig. 9. To produce the executable code, simply run "*make all*".

4.2.2. Implementing a new application

A new application is implemented by coding the problem-dependent components. If the problem is not in the form of a minimization problem, it should be transformed into one using techniques discussed in Sec. 2. The information needed in implementing an application includes the following. a) *Problem Definition.* Problem-dependent data structures and global variables need to be declared, and their initialization routines defined. b) *Assessment Routines.* The routines for evaluating feasible and partial solutions during the course of the search have to be defined. For optimization problems, the upper- and lower-bound functions have to be defined as well. c) *Problem-Instance Generator.* This is used to generate a problem instance based on given problem size and a surrogate, which is a seed used to reset the random number generator. d) *Search-Node Initialization Routines.* These routines allocate search nodes by calling search-node allocation facility built in ISE, initialize its problem-dependent part, and set up links inside it to problem-dependent size-dependent fields. e) *Problem-Dependent Search Elements.* These include the initialization of problem-dependent components of the search environment, feasibility and infeasibility tests, and branching resolution.

4.2.3. Using ISE in resource-constrained search

In this section, we describe our experience in using ISE to experiment with search algorithms in resource-constrained applications, especially for real-time systems. For a user, the command line and output files are the only interfaces to ISE; therefore, we will focus our discussion on them.

```
# application
TARGET_PROB = TSP
EXEC_CODE = ise.tsp
# flags
CC = cc -DDEBUG -DSYM_TSP -DLOWB_GUIDANCE
LoadFlags = -lm
# directories
ISE = ise-src
APPL = $(ISE)/solver/ats.pgm
# '.h' files
DotHFiles = $(ISE)/include/limits.h $(APPL)/*.h $(ISE)/include/*.h
BinFiles = algorithm.o include.o interface.o kernel.o open.o
           primitive.o problem.o
# commands
all: $(DotHFiles) $(BinFiles)
$(CC) $(BinFiles) $(LoadFlags) -o $(EXEC_CODE)
algorithm.o: $(DotHFiles) $(ISE)/algorithm/*.c
cat $(DotHFiles) $(ALG)/*.c | $(CC) -c -o algorithm.o
include.o: $(DotHFiles) $(ISE)/include/*.c
cat $(DotHFiles) $(ISE)/include/*.c | $(CC) -c -o include.o
interface.o: $(DotHFiles) $(ISE)/interface/*.c
cat $(DotHFiles) $(ISE)/interface/*.c | $(CC) -c -o interface.o
kernel.o: $(DotHFiles) $(ISE)/kernel/*.c
cat $(DotHFiles) $(ISE)/kernel/*.c | $(CC) -c -o kernel.o
open.o: $(DotHFiles) $(ISE)/open/*.c
cat $(DotHFiles) $(ISE)/open/*.c | $(CC) -c -o open.o
primitive.o: $(DotHFiles) $(ISE)/primitive/*.c
cat $(DotHFiles) $(ISE)/primitive/*.c | $(CC) -c -o primitive.o
problem.o: $(DotHFiles) $(ISE)/*.c
cat $(DotHFiles) $(APPL)/*.c | $(CC) -c -o problem.o
```

Fig. 9. The makefile in ISE for generating the object code for solving the symmetric traveling salesperson problem.

Assume ISE is compiled for the symmetric traveling salesperson problem, and we wish to solve it using IDA*. An example of the command-line input is

```
ise.tsp :prob 1 20 1 :search dfs ida :pe 1 :pf rt st :io cdo es rwl rtp
stp
```

The problem-specific information is specified in `:prob`; in this case, it says that one TSP instance with 20 cities is to be generated using 1 as the random seed. The search algorithm is specified in `:search`; it says that the primitive search is a depth-first search, with IDA* as the meta-control. The search process is to be run on a single processor, as specified in `:pe`. The profiling mechanisms are specified

in `:pf`, requesting `rt`, the profiling of run-time approximation degree versus time, and `st`, the profiling of memory usage versus time. The custom-designed output is in file `cdo`, the explanatory summary is in file `es`, the row-wise listing is in file `rwl`, and the profiling outputs are in files `rtp` for profiles of run-time approximation degree versus time, and `stp` for profiles of memory usage versus time. The row-wise listing is used for inter-search-process profiling, while the profiling outputs are for intra-search-process profiling.

Figure 10 compares the intra-search-process profile for solving a 20-city symmetric TSP by A* and the corresponding inter-search-process profile of IDA*. Figure 10a shows the decrease in run-time approximation degree during the course of solving a 20-city symmetric TSP by A* and IDA* respectively. It is interesting to note that IDA* achieves good run-time approximation degrees in the early stage, while A* achieves better ones most of the time (note that the time scale is logarithmic). This happens because IDA* goes deep into the search tree and may find a good solution earlier; however, the global lower bound achieved by IDA* does not improve as quickly as that of A*. Figure 10b shows the profiles of memory usage versus time during the course of solving the same TSP. It shows that IDA* uses negligible space as compared to that of A*. Figure 10c shows the profiles of cumulative space-time product versus time for solving the same TSP. Both A* and IDA* require exponential amounts of cumulative space-time product due to the exponential amount of execution time.

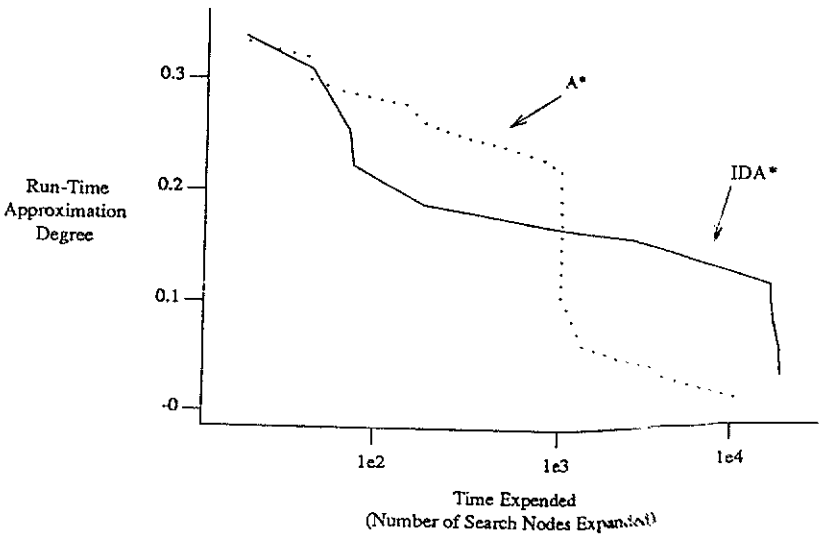
Figure 11 shows the intra-search-process profile of a guided depth-first search as compared to the inter-search–process profile for a version of the RTS[3]. We found that by using meta-control, RTS achieves better approximation degree as compared to a straightforward search algorithm without meta-control.

5. Concluding Remarks

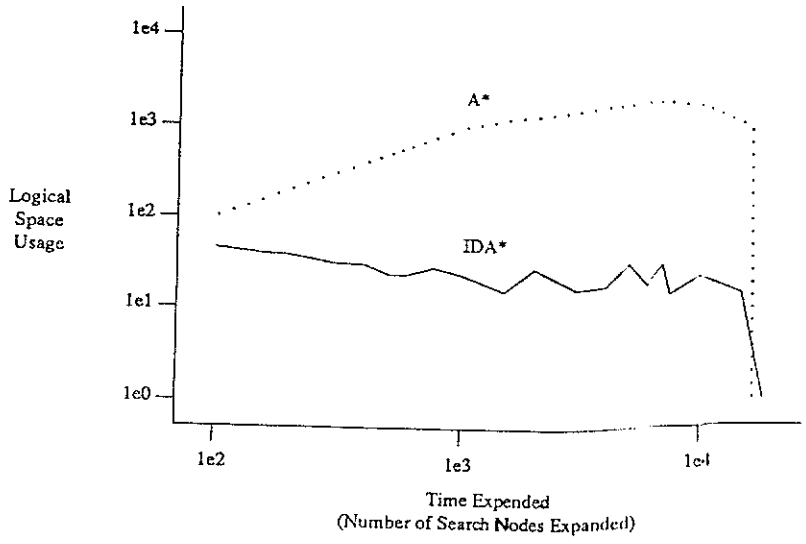
In this paper, we model searches with meta-control as a hierarchy of HSPs (or Hierarchical Search Processes), each represented using context-sensitive and context-free grammars and further classified into problem-independent and problem-dependent components. The new model facilitates the formulation of iterative searches, recursive searches, and resource-constrained searches, especially those with meta-control. It further simplifies the programming of search algorithms and applications requiring searches, as software routines developed for one application or search algorithm can be shared by others. We present ISE, a tool we developed for supporting the programming and experimentation of search processes. Our tool can significantly reduce programming efforts and can serve as a testbed for various search algorithms and applications.

Acknowledgments

This research was partially supported by National Aeronautics and Space Administration under contract NCC 2-481 and by National Science Foundation under grant MIP 88-10584.

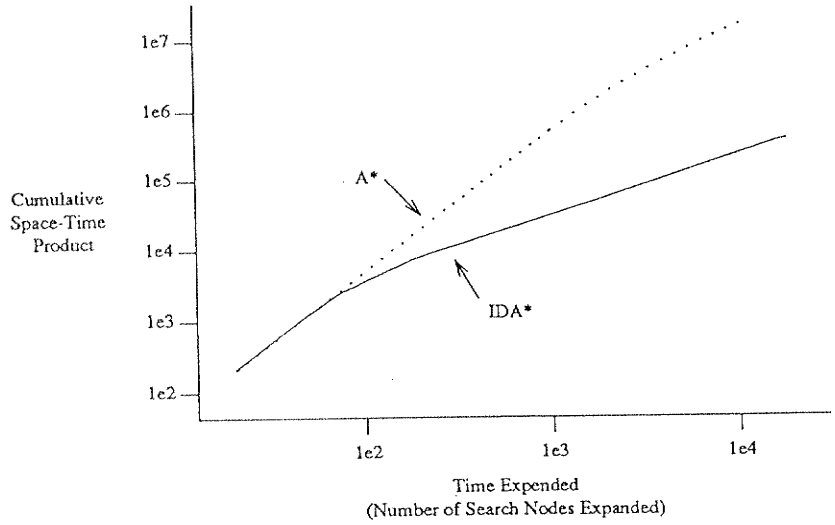


(a) Run-time approximation degree versus time expended



(b) Memory usage versus time expended

Fig. 10. Intra-search-process profiles of A* as compared to inter-search-process profiles of IDA* during the course of solving a 20-city symmetric TSP instance with random seed 1.



(c) Cumulative space-time product versus time expended

Fig. 10. (*Continued*)

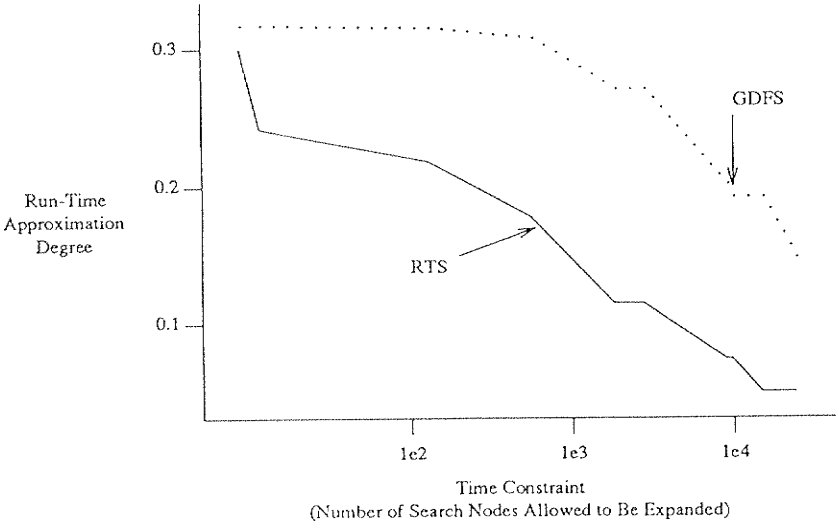


Fig. 11. Intra-search-process profiles of guided depth-first search as compared to inter-search-process profiles of RTS during the course of solving a 20-city symmetric TSP instance with random seed 2.

References

[1] P. P. Chakrabarti, S. Ghose, A. Acharya and S. C. de Sarkar, "Heuristic search in restricted memory", *Artificial Intelligence* **41** (1989) 197–221.

[2] L.-C. Chu, *ISE - An Integrated Search Environment: The Manual*, Tech. Rep. CRHC-92-1, Center for Reliable and High Performance Computing, Coordinated Science Laboratory, University of Illinois, Urbana, IL, Jan. 1992.

[3] L. C. Chu and B. W. Wah, "Optimization in real time", *Proc. Real Time Systems Symposium*, IEEE, Nov. 1991.

[4] L.-C. Chu and B. W. Wah, "Band search: an efficient alternative to guided depth - first search", *Proc. Int'l Conf. on Tools for Artificial Intelligence*, IEEE, Nov. 1992.

[5] S. Gnesi, A. Martelli and U. Montanari, "Dynamic programming as graph searching", *J. ACM* **28** (1982) 737–751.

[6] P. Helman, "An algebra for search problems and their solutions", *Search in Artificial Intelligence*, eds. L. Kanal and V. Kumar (Springer-Verlag, New York, 1988) pp. 28–90.

[7] R. Karp and M. Held, "Finite state processes and dynamic programming", *SIAM J. on Appl. Math.* **15** (1967) 693–718.

[8] R. E. Korf, "Depth-first iterative deepening: an optimal admissible tree search", *Artificial Intelligence* **27** (North-Holland, 1985) 97–109.

[9] R. E. Korf, "Real-time heuristic search", *Artificial Intelligence* **42** (Elsevier Science Publishers, 1990) 189–211.

[10] V. Kumar and L. N. Kanal, "A general branch and bound formulation for understanding and synthesizing and/or tree search procedures", *Artificial Intelligence* **21**, No. 1-2 (North-Holland, 1983) 179–198.

[11] V. Kumar and L. Kanal, "The CDP: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound", *Search in Artificial Intelligence*, eds. L. Kanal and V. Kumar (Springer-Verlag, New York, 1988) pp. 1–27.

[12] E. L. Lawler and D. W. Wood, "Branch and bound methods: a survey", *Operations Research* **14** (ORSA, 1966) 699–719.

[13] V. R. Lesser, J. Pavlin and E. Durfee, "Approximate processing in real-time problem solving", *AI Magazine*, AAAI (Spring 1988) 49–61.

[14] G.-L. Li and B. W. Wah, "Parallel iterative refining A*: an efficient search scheme for solving combinatorial optimization problems", *Proc. Int'l Conf. on Parallel Processing*, St. Charles, IL, Aug. 12–16, 1991.

[15] N. Rao Vempaty, V. Kumar and R. E. Korf, "Depth-first vs best-first search", *Proc. National Conf. on Artificial Intelligence*, AAAI, Anaheim, CA, July 1991.

[16] B. W. Wah and L.-C. Chu, "TCA*–A time-constrained approximate A* search algorithm", *Proc. Int'l Workshop on Tools for Artificial Intelligence*, IEEE (Nov. 1990) pp. 314–320.

[17] B. W. Wah and L.-C. Chu, "TCGD: A time-constrained approximate guided depth-first serach algorithm", *Proc. Int'l Computer Symposium*, Taiwan, China (Dec. 1990) pp. 507–516.

[18] B. W. Wah, *MIDA*: An IDA* Search with Dynamic Control*, Research Report CRHC-91-09, Center for Reliable and High Performance Computing, Coordinated Science Laboratory, Univ. of Illinois, Urbana, IL 61801, April 1991.