ISE -- AN INTEGRATED SEARCH ENVIRONMENT THE MANUAL

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
chu@aquinas.csl.uiuc.edu

ABSTRACT

In this manual, we describe the software package ISE (acronym for *Integrated Search Environment*), a tool that implements hierarchical searches with meta-control. ISE actually is a collection of problem-independent routines to support search processes. Mainly, these routines are core routines for solving a search problem and they handle the control of searches and maintain the statistics related to searches. By separating the problem-dependent and problem-independent parts in ISE, new search methods can be implemented by calling existing methods and they can be developed easily by coding the meta-control. Further, new applications can be developed by only coding the problem-dependent parts. Potential users of ISE would be designers of new application solvers and new search algorithms, and users of experimenting them. ISE is designed to be user-friendly and information rich. In this manual, the organization of ISE is described and some sample runs are also shown.

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

Technical Report CRHC-92-1.

1. INTRODUCTION

This manual describes the usage of our tool, ISE (*Integrated Search Environment*), that facilitates the coding of solving search problems. ISE is the implementation of the hierarchical search processes proposed by Wah [10]. ISE actually is a collection of problem-independent routines to support search solving. Mainly, these routines are *core* routines of solving a search problem, since they handle the control of searches and maintain the statistics related to searches. Potential users of ISE would be designers of new application solvers, designers of new search algorithm, and persons that use search algorithms and application solvers.

New application solvers only need to implement the problem-dependent part and need to interface some parameters related to problem-related characteristics, like the data type of cost values and the problem-dependent part of search nodes, such that the core routines can do things right. Once a new application solver is implemented in ISE, then this application problem can be solved by any of search strategies and search algorithms implemented in ISE in the past or in the future.

New search algorithm designers also can implement their new search algorithms by manipulating builtin search primitives provided by ISE. A search primitive is the most basic search method such as best-first search and depth-first search.

Application users will find the friendliness of ISE and can reconfigurate the search processes via the command line, like search strategy, search algorithm, and profiling status.

In this report, the *traveling salesperson problem* (TSP) and IDA* search algorithm [4] will be used as running examples to illustrate the procedure of coding new application problem solvers and new search algorithms, respectively, on ISE.

This manual is organized as follows. Section 2 briefly describes the architecture of ISE. Section 3 describes WISE core routines. Section 4 describes the development of application problem solvers. Section 5 describes the interface between ISE and new application solvers. Section 6 describes the procedure of implementing new search algorithms on ISE. Section 7 describes sample runs and sample profiles. Finally, section 8 draws the conclusion.

2. ISE ARCHITECTURE

The ISE architecture can be illustrated in Figure 2.1. Potential users can add new search algorithms and new application solvers into ISE with minimal interfacing and minimal efforts in coding. Further, application users can solve the search problem by using any search algorithm in ISE simply via the parameters in the command line. In this sense, ISE is an open software and it can be expanded by adding new application solvers and new search algorithms.

Physically, ISE is a collection of routines about search primitives, search control, statistics, and profiling mechanism. They are stored over several directories: `include/`, `primitive/`, `algorithm/`, `kernel/`, `open/`, `interface/`, and `solver/`.

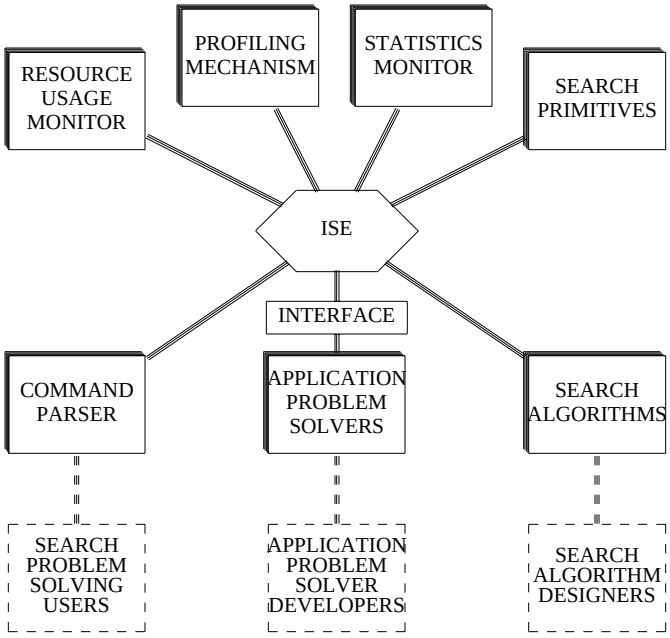

Figure 2.1 ISE Architecture.

Directory `include/` contains all necessary macros, constants, data structures, and declarations of global variables. Directory `primitive/` contains all the search primitives which is the building block of all search algorithms in ISE. The basic primitives currently implemented are

`bfs_primitive`: best-first search primitive.
`dfs_primitive`: depth-first search primitive.
`gdfs_primitive`: guided depth-first search primitive.
`gbb_primitive`: generic branch-and-bound search primitive.
`band_primitive`: band search primitive.

Directory `alogirhtm/` contains all the routines of search algorithms. Default search algorithms means pure BFS, DFS, GDFS, and B&B. In fact, BFS and GDFS can be represented by B&B with best-first and depth-first selection functions, respectively. However, the memory behavior of these searches are entirely different; therefore, they are intended to be seperated. Other search algorithms are implemented by manipulating search primitives or algorithms.

Directory `kernel/` contains all *kernel* routines about search maintenances, resource usage monitoring, resource constraint checking, and statistics monitoring.

Directory `open/` contains routines which dynamically allocate chunks of space for search nodes. The deallocations are done by pooling the deallocated space for future allocations. Also, this directory contains the manager of active lists in the search. Note that the active lists may be different for different search primitives due to efficiency reasons.

Directory `interface/` contains all the routines which parse context-free, user-friendly command lines and generate particular statistics specified by users.

All the application solvers are stored in directory `solver/`. The problem-dependent routines for one specific problem are kept in one directory, for example `solver/ats.pgm/` for the TSP, which contains several small files including routines of search node allocation, search node evaluation, and search node decomposition.

The *search primitives* are the basic search units. Currently, best-first, depth-first, guided depth-first, generic branch-and-bound search, and band search primitives are provided in ISE. From now on, let ‘‘bfs’’, ‘‘dfs’’, ‘‘gdfs’’, ‘‘gbb’’, and ‘‘bs’’ denote best-first, depth-first, guided depth-first, generic branch-and-bound search, and band search, respectively.

Before a search primitive can be called, some information called *search parameters* must be ready, and they include

(1) *Resource Constraints.* Constraints of time, space, and cumulative space-time (CST) cost can be specified such that the search will be terminated when any of these hard constraints is violated.

(2) *Degree of Approximation.* The expected degree of approximation of the solution can be used as a parameter to the search primitives. The degree of approximation is in the range of $[0, \infty)$. The solution is called the optimal solution if the degree of approximation is zero; otherwise, the solution is called α-approximately semi-optimal solution, if the degree of approximation is α.

(3) *Statistics Status.* The search process can be measured *logically* or *physically*. The measurement includes time, space, and CST cost. The physical time is measured by the timer provided by the operating system. The physical space is measured by the number of words in storage used for storing search nodes. The physical CST is the cumulative product of physical space and physical time. The logical time is measured by the number of search nodes expanded or the number of search nodes generated according to user’s specification. The logical space is measured by the number of active search nodes. The logical CST is the cumulative product of logical space and logical time.

(4) *Profiling Status.* The search process can be profiled at run time. The profiling status gives the information about what kind of profiling needs to be done. The profiling covers time, space, CST cost, and degree of approximation.

(5) *Algorithm-Specific Constraints.* The algorithm-specific constraints differ from the hard resource constraints in the way that when the algorithm-specific constraints are violated only the search primitive is terminated, while the entire search algorithm may be terminated if any of the hard resource constraints is violated. In practice, the algorithm-specific constraint may be the expected optimal value, *e.g.* threshold in IDA* [4].

A search algorithm can be defined recursively by either a sequence of search primitives or search algorithms. A sequence is a tuple of one or more ordered items. A search algorithm is said *in ISE format* if it is represented by a sequence of search primitives or search algorithms. For example, IDA* [4] can be represented by a sequence of ‘‘dfs’’ primitives with proper setting of resource constraints and algorithm-specific constraints.

By feeding proper parameters, we can do profiling *intra* search primitives or *inter* search primitives. We can implement new search algorithms by simply manipulating search primitives. One powerful support in ISE is that any search algorithm of hierarchical fashion can be easily implemented by manipulating builtin search algorithms and search primitives. Several of the famous search algorithms are represented here in ISE format as examples of manipulating search primitives and search algorithms.

A* = <bfs>
DFS = <dfs>
IDA = <dfs-1, dfs-2, ..., dfs-n>
sTCA* = <bfs-1, bfs-2, ..., bfs-n>
pTCA* = <bfs> ; with profiling intra search primitive.
dTCA* = <bfs> ; with profiling intra search primitive.
sTCGD* = <gdfs-1, gdfs-2, ..., gdfs-n>
pTCGD* = <sTCGD*, gdfs> ; with profiling inter search primitives.
Beam Search = <gbb> ; by properly defining selection and pruning.
Hill-Climbing Search = <gbb> ; by properly defining selection and pruning.

3. ISE CORE ROUTINES

In this section, the problem independent part of ISE is described. The core routines are kept in several function-specific directories. Roughly, these core routines can be classified into several categories: definitions and declarations (`include/`), search primitives (`primitive/`), search algorithms (`algorithm/`), ISE kernel (`kernel/`), memory management (`open/`), IO transducer (`interface/`). These categories are described following.

3.1. Definitions and Declarations

The definitions of data structures are put in `include/define.h` such that all routines can access these definitions. The problem-independent part of the search node is also defined in this file; while the problem-dependent part is defined by users. The Makefile will automatically include the definition of the problem-dependent part and make the definition of node structure complete. The declarations of all global variables and the routines of initializing data structures are put in `include/var.c`. The corresponding external declarations are in `include/var.h`. The limiting constants, *e.g.* maximum integer, maximum long, and maximum floating-point number, are defined in `include/limits.h`, which is used to patch the systems that do not support complete limiting constants, *e.g.* Sun 3/60 workstation. Some global variables used for debugging are defined in `include/debug.h` and declared in `include/debug.c`.

There is a trick for all definition and declaration routines such that all of them will be compiled once. The trick is setting a compiler directive to check reentrance.

```
#ifndef __define_h_
#define __define_h_
<body of define.h>
#endif  __define_h_
```

3.2. Search Primitives

The search primitives supported in ISE include ‘‘bfs’’, ‘‘dfs’’, ‘‘gdfs’’, ‘‘gbb’’, and ‘‘bs’’ primitives. They are located in `primitve/bfs.c`, `primitive/dfs.c`, `primitive/gdfs.c`, `primitive/bb.c`, and `primitive/band.c`, respectively. The input arguments to these search primitives are a pointer to a search structure, a pointer to its parent’s search structure, and a message consisting of search controls and profiling status.

The search structure, see `search_ { ... }`, includes resource constraints, cost data type, measurement system, statistics status, algorithm-specific constraints, degree of approximation, and some algorithm-specific information.

The resource constraints are used to limit the resource usage of the search in a way that the search is terminated when any of the resource constraints is violated. If the search is at the top level, *i.e. root search*, then the entire search process is terminated when any of the resource constraints is violated. However, if the search is at the intermediate or leaf level of the hierarchy of searches, *i.e.* at the root, then only the particular search is terminated when any of its resource constraints is violated.

The cost data type indicate the data type of the solution value, *e.g.* integer or float. The measurement system indicates that time, space, and CST cost are either logical or physical. In fact, both physical and logical measures are taken in ISE.

Statistics status keeps track of statistics of search performance like number of search node generated, number of search node expanded, number of search nodes pruned, and number of feasible-solution search nodes. These statistics are very important, since they can be used to determine the performance of a particular search algorithm. The parent’s search structure is used only for the statistics tracing.

The algorithm-specific constraints provide another way to terminate the search prematurely. Typically, the expected optimal solution value is one of the soft constraints, which is used in IDA* [4], *i.e.* the threshold.

The degree of approximation is used to solve the problem approximately by pruning more nodes that might lead to the optimal solution. The approximation means the solution is complete but is not optimal, instead of incomplete or partial solution.

All the search primitives have an algorithm-specific entry inside their routines, which allows the search algorithm designers to do some accounting, maintenance, and search process adapting.

The search message has a control flag indicating this particular search primitive should be started freshly or simply resumed. This allows to resume an old search terminated prematurely.

The algorithm-specific entry, the capability of resumption, and the algorithm-specific constraints allow the search algorithm designers to have the freedom to manipulate different styles of search algorithms.

3.3. Search Algorithms

The search algorithm designers can put the source codes of their algorithms to the directory `algorithm/`. Currently existing algorithms include pure BFS, DFS, GDFS, general B&B [5], and band search [2], as well as TCA* (sTCA*, pTCA*, and dTCA*) [8] and TCGD (sTCGD and pTCGD) [9], iterative deepening A* (IDA*) [4], iterative refining A* (IRA*) [6], DFS* [7], real-time search (RTS) [1], and Lawler and Wood's time-constrained approximation search [5].

All these search algorithms can be defined as a sequence of either search primitives or search algorithms. For example, IDA* is a sequence of ''dfs'' primitives, and pTCGD is a sTCGD algorithm (with profiling inter search primitives) followed by a ''gdfs'' primitive. Through the support of search primitives and profiling mechanisms, the search algorithm designers will find that it is very easy to implement their search algorithms by manipulating the search primitives and profiling mechanisms.

3.4. ISE Kernel

The kernel includes the main routine and all the routines about tracing statistics, profiling, and supporting utilities. The statistics include the performance of the search process and the measures of the resource usages. The profiling can be either intra search primitive or inter search primitives. If the profiling is intra search primitive, then the search message fed to the search primitive must have a signal indicating that. If the profiling is inter search primitives, then the profiling routines can be called between two contiguous search primitives. The supporting utilities include a wide spectrum of small routines which make coding easy and data structures abstract.

3.5. Memory Management

The memory management includes the allocation and deallocation of search nodes as well as the handling of the active lists of search nodes.

The allocation and deallocation of search nodes should match the computer architecture and hierarchical memory organization such that the search process will not suffer from frequent swapping or thrashing due to virtual memory faults. In ISE, the search nodes are allocated chunks by chunks, where the chunk size `AllocationSize` is defined by the application solver, because the size of a search node may vary over several orders of magnitude for different applications. In this way, the search nodes have high probability that they are located nearby such that they will not spread over the entire virtual memory. Further, the deallocated search nodes are kept in a linked list for future allocations rather than released to the operating system. In this way, the spatial locality and temporal locality during the search can be reserved.

The active list is a conceptual term, because it is physically different for different search primitives. For the ''bfs'' primitive, the active list should be a B+ tree, which allows logarithmic time to insert or delete a search node in the B+ tree as well as provides easy locating the search nodes whose lower bounds no less than a certain value such that pruning in the active list can be done more easily. However, the general ''bfs'' primitive should allow a set of keys instead of a single key, therefore, the ''bfs''

primitive should use a linked list for its active list due to difficulty in pruning by bounding. If the compiler option `BPLUS_TREE` is defined, then the B+ tree is used; otherwise, the linked list is used.

For both ‘‘gdfs’’ and ‘‘gbb’’ primitives, the active list is simply a linked list. The reason is that the number of search nodes in the active list of a ‘‘gdfs’’ primitive is bounded by the product of the maximum branching degree and the maximum depth of the search tree. The reason of using a linked list for the ‘‘gbb’’ primitive is that the active list is ordered by a certain selection function defined by the application solver and the pruning function is also defined by the application solver. Both functions are application problem dependent so that the B+ tree will make pruning very complex. Note that pruning is not necessarily to be bounding, and it can be dominance and approximation. Therefore, the simple linked list is used for the ‘‘gbb’’ primitive.

For the ‘‘dfs’’ primitive, the active list is simply a stack whose size is bounded by the depth of the search tree. The ‘‘dfs’’ primitive is actually a backtracking search and all search nodes are partially expanded instead of being fully expanded as in other search primitives; therefore, a stack must be used for backtracking instead of linked list or B+ tree.

3.6. IO Transducer

The IO transducer consists of the command-line processor and the versatile output generator.

3.6.1. Command Line Processor

The command-line processor in `interface/command.c` parses the command line and transforms the command parameters into the internal format of search controls. Application users can define which search strategies and algorithms are to be used for solving the application. Further, users also can define how many sample problems are to be solved as well as the profiling status and the resource constraints via the command line. The typical command line is following.

```
search :prob iter size seed
       [:virtual | :real]
       [:report [report-file]]
       [:search bfs   [default | alg] |
        :search gdfs  [default | alg] |
        :search dfs   [default | alg] |
        :search band  [default | alg]]
       [:constr time [space [cst]]]
       [:approx approx]
       [:dbg dbg]
       [:io stat [summary [graph-ready [rt-pf [st-pf]]]]]
       [:pf rt [st]]
       [:param num_params p-1 p-2 ... p-n]
```

where '[]' means the parameters are optional and Parameter `:prob` must be specified; otherwise, ISE cannot know what the problem size is, how many sample problems to solve, and how to generate the sample problems.

The measurement of resources is either `:virtual` or `:real`. The former indicates logical measurement and the latter does physical measurement. The `:report` option provides an entry for monitoring the algorithm-specific performance, eg. in IDA* the performance of each iteration can be output to the file specified in `:report`. The `:search bfs default` defines that the search primitive is `bfs` and the search algorithm is `default`, that means that the search algorithm only contains a single search primitive. The `:prob iter size seed` defines that the number of sample problems is `iter`, the sample problem size is `size`, and the seed to the random number generator is `seed`. The `:constr time space cst` defines that the time constraint is `time`, the space constraint is `space`, and the cumulative space-time product constraint is `cst`. The `:approx` defines the degree of approximation. The `:dbg` defines the debugging mode which allows multiple-level details of debugging information. The `:io` defines the names of the output files, where the first one is for the performance statistics in the list format, the second one is for the summary of performance statistics with more detailed information, the third one is for the graph-ready format of performance statistics, the fourth one is for the run-time profile, and the fifth one is for the space-vs-time profile. The `:pf rt st` defines that the run-time profiling status is `rt` and the space-time profiling status is `st`. If the profiling is desired, then the status is `YES`; otherwise, it is `NO`. The `:param` is for algorithm-specific parameters.

If some of these parameters are not specified in the command line, ISE have default values for them. The default command line is as follows.

```
search :virtual
       :report report
       :search bfs default
       :prob undefined undefined undefined
       :constr unlimited unlimited unlimited
       :approx 0.0
       :dbg 0
       :io stat summary graph-ready rt-pf st-pf
       :pf 0 0
```

3.6.2. Output Generator

The output file generator is in `interface/output.c` which can produce the summary of performance of search processes as well as the performance data in the format which can be used directly by the graph package `grap`. The performance items to be put in the graph-ready format can be specified by the application solver in the file named `output.h` in the application-specific directory.

The performance statistics in the list format are generated by the routines in `interface/result.c` which summarize the performance statistics into a list which can be easily used by a LISP program. To be more specific, the items in this list is equal

to the number of sample problems solved plus 1. Each item is a list of performance statistics of the search processes solving a sample problem. The first item is the averages of the performance statistics over all sample problems. The others are the performance statistics for all sample problems, one for each.

4. APPLICATION DEVELOPMENT

In this section, the development of a new application program will be described. The user must provide several problem-dependent routines. These problem-dependent routines include:
(1) *Problem Definition*: The problem-dependent data structures as well as their initialization routines are defined and the corresponding global variables are declared.
(2) *Bounding Function Routines*: They evaluate upper and lower bounding functions, if applicable.
(3) *Sample Problem Generator*: A routine can generate new sample problems upon the problem size and a random seed given.
(4) *Search Node Management Routines*: A collection of routines can (i) allocate search nodes by calling ISE search node allocation facility, (ii) initialize the problem-dependent part of a search node, and (iii) set up the links inside a search node to the problem-dependent size-dependent region.
(5) *Problem-Dependent Search Components*: Feasibility and infeasibility tests as well as search environment initializations are needed.

Currently, several applications have been implemented in ISE, including symmetric traveling salesperson problem (sTSP), knapsack problem (KS), production planning problem (PP), vertex cover problem (VC), weighted completion time problem (WCT), general resource constrained scheduling problem (GRCS), asymmetric traveling salesperson problem (aTSP), maze problem (Maze), and *N*-puzzle problem (Puzzle). The maze generator in ISE is modified from the X11R4 maze generator [3].

5. INTERFACING

The interface between ISE and application solvers are described in this section. The interface includes definitions of data structures, necessary routines, and specifications of parameters.

5.1. Definitions of Data Structures

Several data structures must be defined by the application problem solver developer, since they are problem-dependent. These include the data type of solution values, the allocation chunk size, the problem-dependent part of search nodes, and the solution structure. Take the TSP as an example. The TSP solver must define `domain`, `AllocationSize`, `PDSI_PART` macro, and `solution_` structure, as we can see in Appendix.

5.2. Necessary Routines

Several problem-dependent routines are required to make ISE work correctly. They include sample problem generation, bounding function routines, feasibility and infeasibility tests, adaptive search node decomposition, root generation, search node allocation, search node initialization, search node linkage, solution buffer management, and solution interpretation.

6. DEVELOPMENT OF NEW ALGORITHM

In this section, the procedure of developing a new search algorithm is described. The procedure is simple and it include the following steps.
(1) Map the new search algorithm into a sequence of search primitives.
(2) Set up the bookkeeping and interfacing between contiguous two search primitives.
(3) Translate the sequence of search primitives and the associated bookkeeping and interfacing into the C language forms ISE supports.

Take IDA* as an example. First, IDA* can be represented as a sequence of ‘‘dfs’’ primitives. Second, design the algorithm which maintains the threshold setting between contiguous ‘‘dfs’’ primitives. Third, translate them into the C language forms ISE supports. The IDA* search algorithm in the ISE format can be found in Appendix.

7. SAMPLE RUNS

In this section, we will show some sample commands and their results.

Assume ISE is compiled for the symmetric TSP by command ‘‘make all’’. Assume we wish to solve a fifteen-city TSP instance using best-first search. Then, the command-line input is

```
ts :prob 1 15 1 :search bfs default
```

The summary result in file `summary` is

```
counting system=virtual
time unlimited
space unlimited
cst unlimited
real time=677
real max space=27280
real cst=1.18474e+07
virtual time=415
virtual max space=155
virtual cst=41221
root approx=0.300167
run-time approx=0
optimal solution value=318.554
cmd-line (or adapted) approx=0
achieved approx=0
```

```
run-time approx=0
```

The above summary is quite self-explanatory. The row-wise result in file `graph-ready` is

```
999999999  415  155  41221  0.300167  0  0  318.554  318.554  1e+20
```

The first item is the time constraint, where ‘‘999999999’’ denotes infinity. The second, third, and fourth items are completion time, maximum space used, cumulative space-time product, respectively. The fifth one is the run-time approximation degree of the root node. The sixth and seventh items are the final run-time approximation degree and the final achieved approximation degree, respectively. Both are zeros because the optimal solution is solved. The eighth and ninth items are the incumbent and the global lower bound, respectively. Both are equal to the optimal solution. The last one is the threshold, where ‘‘1e+20’’ denote the infinite threshold.

If we want to use guided depth-first search for solving the same TSP instance. The command is

```
ts :prob 1 15 1 :search gdfs default
```

The summary file becomes

```
counting system=virtual
time unlimited
space unlimited
cst unlimited
real time=677
real max space=5280
real cst=1.67218e+06
virtual time=421
virtual max space=30
virtual cst=6095
root approx=0.300167
run-time approx=0
optimal solution value=318.554
cmd-line (or adapted) approx=0
achieved approx=0
run-time approx=0
```

The ‘‘graph-ready’’ file is cumulative and now becomes

```
999999999  415  155  41221  0.300167  0  0  318.554  318.554  1e+20
999999999  421  30  6095  0.300167  0  0  318.554  318.554  1e+20
```

If we want to use the band search of bandwidth 5 for solving the same TSP instance. The command is

```
ts :prob 1 15 1 :search band default :param 2 5 F
```

where parameter ‘‘F’’ means the bandwidth is fixed all the time. The summary file becomes

```
counting system=virtual
time unlimited
space unlimited
cst unlimited
real time=663
real max space=17072
real cst=4.56086e+06
virtual time=415
virtual max space=97
virtual cst=16821
root approx=0.300167
run-time approx=0
optimal solution value=318.554
cmd-line (or adapted) approx=0
achieved approx=0
run-time approx=0
```

The ‘‘graph-ready’’ file is cumulative and now becomes

```
999999999  415  155  41221  0.300167  0  0  318.554  318.554  1e+20
999999999  421  30  6095  0.300167  0  0  318.554  318.554  1e+20
999999999  415  97  16821  0.300167  0  0  318.554  318.554  1e+20
```

Note that the third one is band search of bandwidth 5. In many cases, band search (cf. data in the third row) can almost use the same amount of time as best-first search (cf. data in the first row), but in general requires less memory. In fact, band search use bounded amount of memory [2].

8. CONCLUDING REMARKS

ISE is still an evolving software to support the research on design of resource-constrained search algorithms. ISE is aimed to support a wide spectrum of search algorithms and application solvers. Its goal is to move the problem-independent part of application solvers into an integrated kernel such that the integrated kernel is transparent to the designers of either new applications or new search algorithms such that they can be done in less time as possible.

ACKNOWLEDGEMENT

I would like to give my great thanks to Professor Benjamin .W. Wah for his constant and helpful discussions on improving ISE model and structures. I also would like to thank Arthur Ieumwananonthachai for his helpful discussions on coding. This work is

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

REFERENCES

[1] L.-C. Chu and B. W. Wah, ‘‘Optimization in Real Time,’’ *Proc. Real Time Systems Symp.*, pp. 150-159, IEEE, Nov. 1991.

[2] 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*, pp. 154-161, IEEE Computer Society, Nov. 1992.

[3] R. Hess, D. Lemke, and M. Weiss, ‘‘Maze,’’ *X Version 11 Release 4*, July 1991.

[4] R. E. Korf, ‘‘Depth-First Iterative Deepening: An Optimal Admissible Tree Search,’’ *Artificial Intelligence*, vol. 27, pp. 97-109, North-Holland, 1985.

[5] E. L. Lawler and D. W. Wood, ‘‘Branch and Bound Methods: A Survey,’’ *Operations Research*, vol. 14, pp. 699-719, ORSA, 1966.

[6] G.-J. 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*, pp. 608-615, Pennsylvania State Univ. Press, University Park, PA, Aug. 1991.

[7] 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.

[8] B. W. Wah and L.-C. Chu, ‘‘TCA*--A Time-Constrained Approximate A* Search Algorithm,’’ *Proc. Int’l Workshop on Tools for Artificial Intelligence*, pp. 314-320, IEEE, Nov. 1990.

[9] B. W. Wah and L.-C. Chu, ‘‘TCGD: A Time-Constrained Approximate Guided Depth-First Search Algorithm,’’ *Proc. Int’l Computer Symp.*, pp. 507-516, Tsing Hua Univ., Hsinchu, Taiwan, Dec. 1990.

[10] B. W. Wah and L.-C. Chu, ‘‘Combinatorial Search Algorithms with Meta-Control: Modeling and Implementations,’’ *Int’l J. of Artificial Intelligence Tools*, vol. 1, no. 3, pp. 369-397, World Scientific Publishers, Sept. 1992.

APPENDIX

The source code of ISE software is listed as below. The software consists of ‘‘include’’, ‘‘primitive’’, ‘‘algorithm’’, ‘‘kernel’’, ‘‘open’’, ‘‘interface’’, and ‘‘solver’’ directories. The applications include TSP, KS, PP, VC, WCT, GRCS, Maze, and Puzzle. Further, relevant makefiles are also listed.