LEARNING TO COMBINE HEURISTICS TO SOLVE CONSTRAINT SATISFACTION PROBLEMS

by

SMILJANA PETROVIC

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2008

This manuscript has been read and accepted for the
Graduate Faculty in Computer Science in satisfaction of the
dissertation requirement for the degree of Doctor of Philosophy.

Date	Dr. Susan L. Epstein Chair of Examining Committee
Date	Dr. Theodore Brown Executive Officer

Dr. Gary Bloom

Dr. Theodore Brown

Dr. Richard Wallace
Supervision Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

LEARNING TO COMBINE HEURISTICS TO SOLVE CONSTRAINT SATISFACTION PROBLEMS

by

Smiljana Petrovic

Adviser: Professor Susan L. Epstein

Problem solvers have at their disposal many heuristics that may support effective search. The efficacy of these heuristics, however, varies with the problem class. This research is on machine-learning techniques to select appropriately from among a large body of heuristics, and combine them into a weighted mixture that works well on a specific class of constraint satisfaction problems.

The learning used here is a form of self-supervised reinforcement learning. The training instances come from the solver's own (likely imperfect) successful searches. As a result, learning lacks reliable information on how much any individual decision or mutual interaction of heuristics influenced overall search performance.

A pre-existing learner has weights for heuristics learned from the size of the search space explored. The new algorithms proposed and tested here use different ways to gauge and adapt search performance; they learn from the accuracy, intensity, frequency and distribution of heuristics' preferences. Stochastic methods address challenges for that learner on hard problems under limited resources and with large and contradictory set of heuristics.

This work is an important step toward automated problem solving. The resultant solver is effective, robust and self-adaptive. It thereby relieves the user of the burden of providing domain specific knowledge for a solver. Although the experimental work is done on constraint satisfaction problems, the machine learning techniques presented here are general, and therefore applicable to many other domains.

Acknowledgments

I would like to express my sincere gratitude to my Adviser, Dr. Susan L. Epstein for her leadership, support, hard work, devotion, experience, and for guiding my growth as a researcher. She was always available when I needed her help or advice, and enthusiastic about the project and the results. She has inspired my efforts through her own scientific curiosity and has given me extraordinary experiences throughout our work on many fascinating projects.

I would like to thank Dr. Richard Wallace from the Cork Constraint Computation Centre for his interest, insightful comments and suggestions, and constructive criticism that were continuous sources of enhancement of my scientific endeavor. Appreciation also goes out to Dr. Eugene Freuder for contributing valuable insights in the development of this research. I would like to thank Dr. Bloom and Dr. Brown for being my committee members and for their valuable time in reviewing my thesis.

I would also like to thank my fellow PhD students, Xingjian Li, Tiziana Ligorio, and Zhijun Zhang from the CUNY's FORR study group, and Diarmuid Grimes from the Cork Constraint Computation Centre. We shared many discussions and I am looking forward to continue our collaborations.

I would also like to thank Dr. Victor Y. Pan for encouraging me to pursue my PhD research. I thank Professor Germana Glier from Bronx Community College and Mr. Radomir Kovacevic, Mr. Stephen H. Spahn and especially Ms. Raffaella Depero from the Dwight school for giving me opportunity to pursue my professional career as educator and teaching me about "the unique spark of genius" within each student.

I thank my parents, Mirjana and Vladimir, for their support through all my life. Hvala. My appreciation goes to my sister Ljiljana, for her believing in me and always being there when I needed her. Above all, I am grateful to love of my life, my husband Ivan, for all his support, encouragements, understanding, for making me laugh when I need it and for all his love. We are blessed with the joy and happiness our son Momir brought to our life. Momir makes all efforts worthwhile and he is my greatest source of strength and inspiration.

This thesis is dedicated to Momir.

TABLE OF CONTENS

LIST OF FIGURES

LIST OF TABLES

1. Chapter 1

This research develops machine-learning techniques to solve constraint satisfaction problems by selecting and combining heuristics. The premise of this work is that a good combination of heuristics can outperform even the best individual heuristic. The thesis is that introducing randomness and nuances from heuristics' comparative opinions can be exploited to improve learning performance. The techniques presented here are general and applicable to many machine learning applications. The experimental environment used here is *ACE* (the Adaptive Constraint Engine). While it solves problems from a given class, ACE acquires information and uses it to customize a weighted mixture of pre-specified search heuristics for a particular class.

This chapter first describes fundamental concepts in machine learning and introduces constraint satisfaction problems (*CSPs*), including some classes of CSPs and search methods for solving them. That is followed by a brief overview of related research on adaptive solving of CSPs using multiple heuristics, a description of ACE, and challenges for self-supervised learning on hard problems with a large and inconsistent body of heuristics.

Class-inappropriate heuristics may combine to make bad choices, and thereby make it difficult to solve and learn from the problem within a resource limits. Learning from random subsets of heuristics and full restart are stochastic methods that address those issues in Chapter 2 (Petrovic and Epstein, 2006a; Petrovic and Epstein, 2007b; Petrovic and Epstein, In press).

Subsequent chapters consider a heuristic's support for a set of possible choices.

Chapter 3 introduces a new weight-learning algorithm, *RSWL (Relative Support Weight Learning)*, which bases reinforcement on relative support (Petrovic and Epstein, 2006b). It also reports on how relative support influences search (Petrovic and Epstein, 2007a). Chapter 4 discusses the significance and impact of this work, and outlines plans for future work.

1.1. Machine learning

Artificial intelligence (AI) studies agents (e.g., robots or embedded software systems) that perceive their environment and take actions that maximize their computational ability to achieve goals. AI studies *search* (the examination of large numbers of possibilities), *representation* (an agent's knowledge about the world in general, the specific situation in which it must act, and its goals), *inference* (deduction from known facts), *planning* (the generation of a sequence of actions to achieving a goal) and *machine learning* (techniques that improve agent's performance with experience).

Machine learning is particularly important when it is infeasible for people to test all possibilities, when hidden relations must be extracted from large data sets, or when an environment changes over time. In machine learning, given a set of *training instances X*, described by a set of attributes *A,* an agent tries to create a *hypothesis function h* that approximates an unknown *target function f*. We can think of learning a function as search through a hypothesis space. The *inductive learning premise* is that any hypothesis found to approximate the target function well over a sufficiently large set of training instances also approximates the target function well over other unobserved examples. If the hypothesis space is completely unrestricted, no concise result may be possible. It is therefore necessary either to restrict the space to a smaller class of functions (the

restricted hypothesis space bias) or to introduce a preference relation among the hypotheses (a *preference bias*) (Mitchell, 1997).

In *unsupervised learning*, only training instances are available, without target function values (*direct feedback*) for them. *Clustering* partitions a data set into subsets with common characteristics. *Density estimation* and *anomaly detection* estimate the density function of a large population from which randomly-selected instances are observed. *Object completion* predicts the missing attributes of an instance; it can be implemented using clustering and density estimation methods. *Relevant object retrieval* identifies instances in the collection that are most similar to the partially described instance. *Dimensionality reduction* and *data compression* discover structures with a reduced number of relevant attributes (Dietterich, 2003).

In *supervised learning*, each training instance x includes direct feedback $f(x)$. In *classification,* the target function f is discrete-valued, and indicates the value of some property P on the instance. For supervised learning, performance is evaluated on a previously inexperienced set of instances (*testing instances*). Two common performance metrics are the sum of the mean squared difference $h(x)–f(x)$ over the set of training instances and the total number of testing instances incorrectly classified.

There are many learning methods for classification (Mitchell, 1997). *Decision tree learning* generates a decision tree with attributes as nodes, attribute values as branches, and P or *not-P* as leaves (Buntine, 1991). One approach is to prefer the shortest tree having the highest possible information gain on each level from the top. In *Bayesian learning*, hypotheses are assigned prior probabilities and their conditional probabilities are learned from training examples (Friedman, Geiger and Goldszmidt, 1997). In

instance-based learning, a program memorizes training instances and approximates the function value on a test instance by the values of the training instances closest to it (Aha, Kibler and Albert, 1991). *Genetic algorithms* maintain a set of hypotheses and allow the best of them to survive, reproduce and/or combine to produce a new generation of hypotheses (Schmitt, 2001). Connectionist models *(neural networks)* use a set of processing nodes that are interconnected in a network to identify patterns in training data (White, 1989). Signals travel over these nodes, which can be activated by combined weighted inputs. Such a network learns by updating interconnection weights.

Reinforcement learning addresses problems whose goal states can be recognized, but there is no knowledge about how to reach them. Through trial and error, reinforcement learning learns an *optimal policy*, that is, an action to be performed in each state that maximizes the total reward accumulated on a path to the terminal state (Sutton and Barto, 1998). The *value function* V of a policy π assigns to each state x_t the sum of the rewards obtained by following the policy π from x_t to a terminal state x_n:

$$V^{\pi}(x_t) = \sum_{i=t}^{n-1} \gamma^{i-t} r(x_{i+1}) \qquad [1.1]$$

where $r(x_t)$ is the reward received in the state x_t and $\gamma \in [0,1]$ is a *discount factor* that progressively decreases the influence of rewards received in the future. An *optimal value function* is one that in all states is no worse than any other.

Problems modeled as *MDPs* (*Markov decision problems*, where the choice of an action in a given state depends only on that state, not on prior history) are often solved by reinforcement learning. In a *deterministic MDP* (where the successive state is determined by selected action), the optimal policy can be directly extracted from the optimal value

function. The relation between the values of successive states under an optimal value function is given by the system of *Bellman equations* (Bellman, 1957):

$$(\forall\, x_t)\ V(x_t) = r(x_t)+\gamma V(x_{t+1}) \qquad [1.2]$$

In a *non-deterministic MDP*, there is a probability distribution for successor states when a given action is performed. If the distribution of successor states is unknown, a Monte Carlo method is used to statistically update those probabilities during learning.

Reinforcement learning iteratively updates the values of states based on policy evaluations and value iterations. Starting with arbitrary values, a policy is updated by the greedy selection of an action, and values are updated by the system of Bellman equations. To reduce computation, *Q-learning* associates with each *state-action pair* the total return that would accrue if one started from that state, took that action and afterwards followed the policy (Harmon and Harmon, 1997). Different interpolation methods can extend the scope of the method. *Temporal difference learning* extends a system of Bellman equations from a relation between a state and its successors to a weighted relation between a state and more than one future state. Temporal difference learning propagates the influence of rewards faster than value iteration does (Harmon and Harmon, 1997).

In reinforcement learning it is important to balance exploration and exploitation. If decisions are based only on acquired knowledge, search can be narrowed to a limited number of states (*exploitation*). This computationally inexpensive approach finds a sufficiently good policy, but it does not guarantee an optimal policy. *Exploration* directs search to states not previously encountered, to ensure that the value function is evaluated for more states. Exploration provides more information about a problem. It is forced by

occasionally choosing actions that are not recommended as the best. This work applies reinforcement learning to constraint satisfaction problems.

1.2. Constraint Satisfaction Problems

A *constraint satisfaction problem* P is a triple (X, D, C), where X is a finite set of *variables* $\{X_1, X_2,..., X_n\}$ and D is a function that maps each X_i in X to its *domain* D_{Xi} of *possible values* of X_i. C is a finite (possibly empty) set of *constraints* $\{C_1, C_2,..., C_k\}$, where each constraint C_j is a relation on some subset of variables in X that restricts the values those variables can simultaneously take. A *solution* to a CSP is an assignment of values to all the variables so that no constraint is violated. To solve a CSP may be to find any solution, to find all solutions, or to find an optimal solution, where optimality is defined according to some cost function. A CSP may be *over-constrained* (have no solution) or *under-constrained* (have more than one solution).

The *arity* of a constraint is the number of variables involved. *Binary constraints* are relations between two variables. In a *binary CSP*, all constraints have arity no greater than two. *Equivalent CSPs* have the same set of solutions. Any CSP can be transformed into an equivalent binary CSP. A binary CSP can be represented as a *constraint graph*, where vertices correspond to the variables (labeled by their domains), and each edge indicates the existence of a constraint between its respective variables (labeled with consistent pairs of values). Two variables with an edge between them are *neighbors*.

Many problems may be expressed as CSPs: scheduling problems, satisfiability (*SAT*) problems, graph-coloring problems, and Huffman-Clowes scene labeling problems (Do and Kambhampati, 2001; Lamb and Bandopadhay, 1993; Prestwich, 2008; Sadeh

and Fox, 1996; Walsh, 1999). Search for a solution to a CSP is an NP-complete problem (e.g., 3-SAT and graph coloring problems are NP-complete). Nonetheless, many subclasses with restricted domains, constraints or structure are tractable (e.g., 2-SAT and binary acyclic problems) (Dechter, 2003; Freuder, 1982; Papadimitriou, 1994).

1.2.1. CSP Classes

A *CSP class* here is a set of CSPs with the same characterization. For example, binary CSPs in *model B* are characterized by $<n, m, d, t>$, where n is the number of variables, m the maximum domain size, d the *density* (fraction of edges present out of the $n(n-1)/2$ possible edges) and t the *tightness* (fraction of possible value pairs that a constraint excludes) (Gomes, Fernandez, Selman, et al., 2004).

A class can also mandate some non-random structure on its problems. For example, a *composed problem* consists of a subgraph (called its *central component*) loosely joined to one or more subgraphs (called *satellites*) (Aardal, Hoesel, Koster, et al., 2003). There are no links between satellites. Figure 1.1 illustrates a composed problem with two satellites. The number of variables n, maximum domain size m, density d, and

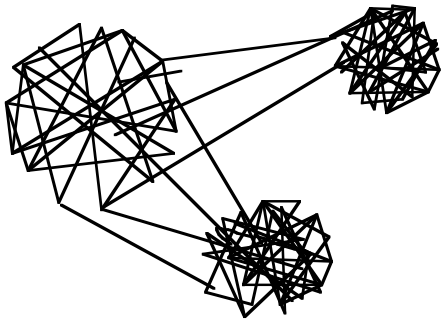

Figure 1.1: A composed problem with two satellites.

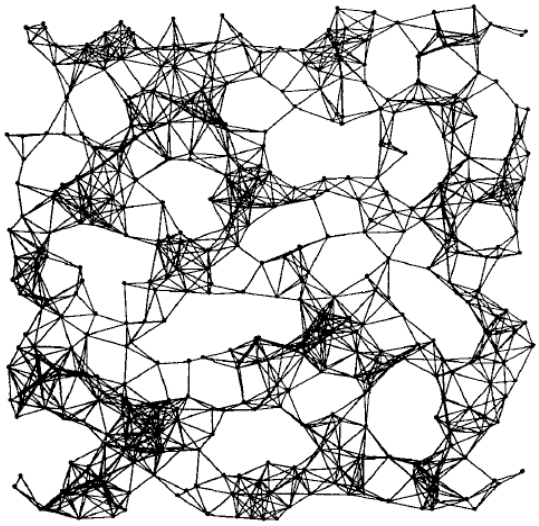

Figure 1.2: Geometric graph from (Johnson, Aragon, McGeoch, et al., 1989)

tightness *t* of the central component and its satellites are specified separately, as are the density and tightness of the links between them. A class of composed graphs is specified here as $<n, m, d, t> s <n', m', d', t'> <d_k, t_k>$ where the first tuple describes the model B central component, *s* is the number of satellites, the second tuple describes the model B satellites, and $<d_k, t_k>$ are the density and tightness of the links between each satellite and the central component, respectively.

A random *geometric graph* $<n, D>$ has *n* vertices, each represented by a random point in the unit square (Johnson, Aragon, McGeoch, et al., 1989). There is an edge between two vertices if and only if their (Euclidean) distance is no larger than *D*. A class of random *geometric CSPs* $<n, D, d, t>$ is based on a set of random geometric graphs $<n, D>$. In $<n, D, d, t>$, the variables represent random points, and constraints are on variables corresponding to points close to each other. Additional edges ensure that the graph is connected. Density and tightness are given by the parameters *d* and *t*, respectively. Figure 1.2 illustrates a geometric graph with 500 variables.

1.2.2. Search methods

There are two broad classes of search methods for solving CSPs: global search and local search. *Global search* methods systematically traverse the space of *partial instantiations* (assignments of values to a subset of the variables). Global search is both *correct* (finds only valid solutions) and *complete* (finds all solutions given enough resources). *Local search* methods stochastically explore the space of *full instantiations* (assignments of values to all of the variables). Local search moves from one full instantiation to another. Although local search is not complete, it is often very efficient and can find a solution very quickly. This work addresses global search.

Figure 1.3 is a high-level global search algorithm. It incrementally extends a partial instantiation with an assignment that does not violate any constraint. If no such an assignment exists, the current instantiation cannot be extended to a solution and global search *backtracks*, that is, retracts some value assignments. *Chronological backtracking* withdraws the most recent value assignment(s). A *propagation method* prunes the search space by early detection of *inconsistencies* (assignments that violate one or more constraints). Values for two variables with a common constraint *support* one another if their simultaneous assignment does not violate any constraint. *Forward checking (FC)* is

Global Search
Until the problem is solved
    Select unvalued variable v
    Select a value d for variable v from v's domain D_v
    Correct the domains of all unvalued variables **propagate**
    Unless the domains of all unvalued variables are non-empty
        return to a previous alternative value **backtrack**

Figure 1.3: Global search to find a single solution to CSP.

a propagation method that reduces the domains of the uninstantiated variables by temporarily removing from the domains of the neighbors of the just-assigned variable values that are not supported by the last assignment (Haralick and Elliott, 1980).

A CSP is *k-consistent* if and only if for every consistent instantiation of *k*-1 variables, there is a value for any k^{th} variable that extends the instantiation to a consistent instantiation of *k* variables (Freuder, 1978). To enforce k-consistency is to transform a CSP into an equivalent *k*-consistent problem. A problem is *strongly k-consistent* if and only if it is *i*-consistent for every $i \leq k$. *Arc-consistency* is strong 2-consistency. An *arc-consistency algorithm (AC)* for binary CSPs removes values that do not have supporting values along each constraint in all other domains. Each value removal may affect previously established supports, so after any variable domain is reduced, each domain of the neighboring variables is rechecked. A *maintained arc consistency (MAC)* algorithm enforces AC after each assignment and after backtracking for each unsuccessful assignment enforces AC once again (Sabin and Freuder, 1997).

The size of the search tree during global search depends in part on the order in which values and variables are selected. Different *variable-ordering heuristics* (rules for selecting the next variable) and *value-ordering heuristics* (rules for selecting the next value to be assigned to that variable) can reduce search or extend it.

1.2.3. Problem difficulty

Some classes of problems are more difficult to solve than others of the same size. Such classes are said to be at the *phase transition* and may be identified by proximity to a critical value of a certain parameter (Cheeseman, Kanefsky and Taylor, 1991). Constrainedness (κ, *kappa*) estimates the overall difficulty of problems in a CSP class as

a function of its parameters (Gent, Prosser and Walsh, 1996). For randomly generated CSPs from model B with <*n, m, d, t*>, constrainedness is estimated by

$$\kappa = \frac{n-1}{2} \cdot d \cdot \log_m(\frac{1}{1-t}) \qquad [1.3]$$

For classes of under-constrained problems, as the number of solutions approaches the number of states, constrainedness approaches 0. As the insolvability of a class of problems becomes easier to demonstrate, constrainedness goes to ∞. A phase transition occurs when there is typically only one solution, where κ is near one. It is important to note that constrainedness estimates the hardness of a class of problems, not of a specific instance in that class.

The time needed to solve random CSP problems in the same class varies greatly from problem to problem (Gomes, Selman, Crato, et al., 2000). This is not necessarily because an individual problem is hard; on another attempt, with a different algorithm, a different heuristic, or even a different random seed, the same problem may be solved quickly. Furthermore, even with an extremely large upper bound on the number of backtracks, a substantial number of CSPs may go unsolved. Standard probability distributions of a random variable often have an exponentially decreasing tail, so that events several standard deviations from the mean are very rare. In contrast, heavy-tailed distributions have tails that decrease according to a power law, that is, the probability $p(X>x) \sim cx^{-\alpha}$, for constants $0<\alpha<2$ and $c>0$. Under a fixed search algorithm, the number of backtracks in model B classes often has a heavy-tailed distribution (Gomes, Selman, Crato, et al., 2000).

1.3. Mixtures of algorithms

The idea that the combined recommendations of multiple human experts can outperform a single expert goes back at least to the Marquis de Condorcet (1745-1794) (Young, 1988). His Jury Theorem asserts that the judgment of a committee of competent experts, each of whom is correct with probability greater than 0.5, is superior to the judgment of any individual expert. Dietterich gives several reasons for preferring a mixture of hypotheses in machine learning (Dietterich, 2000). On limited data, there may be different hypotheses that appear equally accurate. In this case, although one could approximate the unknown true hypothesis by the simplest one, averaging or mixing all of them together can produce a better approximation. Moreover, even if the target function is not representable by any individual hypothesis in the pool, their combination could produce an acceptable representation.

Machine learning methods explore mixtures of *prediction algorithms* (e.g., learning algorithms, classifiers, hypotheses, heuristics, human experts). An *ensemble of classifiers* is a set of classifiers whose individual decisions are combined to classify new examples. *Ensemble methods* create ensemble classifiers. The most popular such algorithm, AdaBoost, seeks a classifier that is better on examples for which the current ensemble's performance is poor (Freund and Schapire, 1996; Schapire, 1990).

Another research area studies how to select and combine experts' predictions. A *mixture of experts algorithm* learns from a sequence of trials how to combine experts' predictions (Kivinen and Warmuth, 1999). In a supervised environment, a trial has three steps: the mixture algorithm receives predictions from each of e experts, makes its own prediction y based on them, and then receives the correct value y'. The objective is to

create a mixture algorithm that minimizes the *loss function* (the distance between y and y'). The performance of such an algorithm is often measured by its *relative loss*: the additional loss compared to the best individual expert. Under the worst-case assumption, mixture of experts algorithms have been proved asymptotically close to the behavior of the best expert (Kivinen and Warmuth, 1999).

There are many theoretical and experimental confirmations of the average case performance superiority of mixtures of classifiers, particularly for decision trees and neural networks (Ali and Pazzani, 1996; Opitz and Shavlik, 1996; Valentini and Masulli, 2002). It has been demonstrated that for sufficiently accurate and diverse classifiers, the accuracy of an ensemble increases with the number of classifiers combined (Hansen and Salamon, 1990).

1.4. Mixtures of algorithms for CSP search

Different algorithms can be combined to solve CSPs. Under *REBA* (Reduced Exceptional Behavior Algorithm), more complex algorithms are applied only to harder problems (Borrett, Tsang and Walsh, 1996). REBA begins search with a simple algorithm, and when there is no indication of progress, switches to a more complex algorithm. If necessary, this process can continue through a pre−specified sequence of complex algorithms. The ranking can be tailored for a given class of problems, and is usually based on the median cost of solution and an algorithm's sensitivity to exceptionally hard problems from the class. In general these algorithms have better worst-case performance, but a higher average cost when applied to classes with many easy problems that could be quickly solved by simpler algorithms.

Algorithm portfolios select a subset of the available algorithms according to some schedule, to be run in parallel (or interleaved on a single processor), with the same priority, until the fastest one solves the problem (Gomes and Selman, 2001). Different methods for schedule selection allow the proportion of CPU time allocated to each heuristic to change over time, and thereby favor more promising algorithms (*dynamic algorithm portfolios)* (Gagliolo and Schmidhuber, 2006) or improve average-case running time relative to the fastest individual solver (Streeter, Golovin and Smith, 2007).

Selecting among different variable-ordering heuristics has been studied for local search. In one approach, a set of heuristics is associated with each constraint (Nareyek, 2004). Each iteration of local search selects a constraint to be adjusted based on some measure of its inconsistency in the current instantiation. The heuristic that makes an adjustment is selected probabilistically, based on its expected benefit (its *utility value*). All utility values are initially equal, and positively or negatively reinforced based on the difference between current total cost and the total cost the last time that constraint was selected.

CLASS discovers good variable-ordering heuristics with a genetic algorithm (Fukunaga, 2002). It begins with a population of randomly generated heuristics, picks two heuristics with probability proportional to some objective function and generates a set of children which are then inserted into the population. Each child replaces a randomly selected member of the population, so that the population remains constant in size. The best heuristic found during the course of the search is returned.

Multi-TAC first generates heuristics specifically for given problems, and then orders them in a list (Minton, Allen, Wolfe, et al., 1995). It starts with an initially empty

list as a parent. Each of the remaining heuristics, attached to the parent one at a time, creates a child. The *utility* of a child is the number of instances solved within a given time limit for each instance, with total time as a tiebreaker. The child with the highest utility is declared the new parent. The process continues recursively until there is no child better than its parent, or all candidates are exhausted. The heuristics in the resulting list are consulted one by one, moving to the next only as a tiebreaker.

1.5. ACE, The Adaptive Constraint Engine

One ambitious approach to CSP solution is based on *FORR* (FOr the Right Reasons), a problem-solving and learning architecture for the development of expertise from multiple heuristics (Epstein, Freuder, Wallace, et al., 2002). FORR is cognitively-oriented; many of its features simulate characteristics of human problem solving (Epstein, 2004a). FORR makes decisions by combining recommendations from procedures called *Advisors*, each of which implements a reason for taking, or not taking, an action. By solving instances of problems from a given class, FORR learns an approach tailored to that class. FORR-based programs have produced expert-level (not necessarily optimal) learners for different domains. *Hoyle* is a FORR-based system for playing two-person, perfect information, finite-board games (Epstein, 1994). *Ariadne* learns to find paths through mazes (Epstein, 1995). *Graph Colorer* solves graph coloring problems (Epstein and Freuder, 2001). *ACE* is a FORR-based program for solving binary CSP problems (Epstein, Freuder, Wallace, et al., 2002).

1.5.1. Search with a mixture of Advisors

ACE performs global search: it alternately selects a variable and then assigns it a value from its domain. ACE propagates after each value assignment. When a value inconsistent with the current partial instantiation is selected, ACE chronologically backtracks. Under *k-way branching,* an alternative value for the last selected variable is then assigned; under *2-way branching*, a (possibly new) variable for assignment is selected.

ACE's Advisors are organized into 3 tiers. If it comments, a tier-1 Advisor is always obeyed. A comment from a tier-1 Advisor either specifies a variable or a value or eliminates one from further consideration in that iteration. An example of a tier-1 Advisor in ACE is *Victory*, which recommends any value from the domain of the last unassigned variable. Tier-1 Advisors are consulted in a pre-specified order.

If no tier-1 Advisor comments, control is passed to the tier-2 Advisors, which detect a subgoal and recommend a sequence of actions to address it. One example of a tier-2 Advisor is *Dendrite,* an Advisor that supports the selection of a tree-like appendage to a cyclic component of more than 5 variables. The tier-1 Advisor *Enforcer* ensures that any subgoal identified by tier 2 is addressed.

Tier-3 Advisors are variable-ordering and value-ordering heuristics. Advisors can be *static* (prespecify an order before search begins) or *dynamic* (dependent on the current state during search). For example, the *Min-static-degree* Advisor selects variables in increasing order of the number of constraints that involve them in the original problem. In contrast, the *Min-dynamic-degree* Advisor selects a variable that is, in the current state, constrained by the fewest uninstantiated variables.

When a decision is passed to tier 3, each tier-3 Advisor comments upon any number of choices. The *strength s(A, c, C)* is the degree of support of the Advisor *A* for the choice $c \in C$. Each tier-3 Advisor's view is based on a descriptive metric. An example of a tier-3 variable metric is *domain/static-degree*, which calculates for each variable the ratio of dynamic domain size to static degree. For each metric, there is a *dual pair* of Advisors, one that favors smaller values for the metric and one that favors larger values (e.g., *Min-domain/static-degree* and *Max-domain/static-degree)*. Typically, one Advisor from each pair is reported as a good heuristic in the CSP literature, but the other may be very successful in some situations, so ACE implements them both. An example of a tier-3 value metric is *product-domain-value*, which associates with each value the product of the resultant domain sizes of its neighbors after propagation. All variable-ordering and value-ordering Advisors used here are described in Appendix A. Two *benchmark Advisors*, one for value ordering and one for variable ordering, generate

Search (p, $\mathcal{A}_{var}$, $\mathcal{A}_{val}$)

Until the problem *p* is solved or the allocated resources are exhausted
  Select an unvalued variable *v*

$$v = \arg\max_{c \in V} \sum_{A \in \mathcal{A}_{var}} w(A) \cdot s(A,c,V)$$ **voting**

  Select an value *d* for variable *v* from *v*'s domain D_v

$$d = \arg\max_{c \in D_v} \sum_{A \in \mathcal{A}_{val}} w(A) \cdot s(A,c,D_v)$$ **voting**

Revise the domains of all unvalued variables **propagation**
  Unless the domains of all unvalued variables are non-empty
    return to a previous alternative value **k-way branching**

Figure 1.4: Global search with k-way branching in ACE to find a single solution to CSP with a weighted mixture of variable Advisors from $\mathcal{A}_{var}$, and value Advisors from $\mathcal{A}_{val}$. *w(A)* is the weight of Advisor *A; s(A, c, C)* is the support of Advisor *A* for choice $c \in C$.

random comments that are excluded from decision-making. The benchmark Advisors provide a lower bound on performance.

All tier-3 Advisors are consulted simultaneously. A decision in tier 3 is made by *weighted voting*, where the strength $s(A, c, C)$ given to choice $c \in C$ by Advisor A is multiplied by the *weight* $w(A)$ of Advisor A. Weighted voting identifies the choice with the greatest sum of weighted strengths from all Advisors. (Ties are broken randomly.) Figure 1.4 is a high-level description of ACE's global search with a weighted mixture of Advisors. Given a class of problems and a set of Advisors, ACE's goal is to learn weights for those Advisors so that the decisions supported by the largest weighted combination of strengths lead to effective search.

1.5.2. Learning from search experience

ACE does a form of reinforcement learning. The only information available to it comes from its limited experience as it finds one solution to a problem, and estimates itself extent of the search effort. This approach is problematic for several reasons. There is no

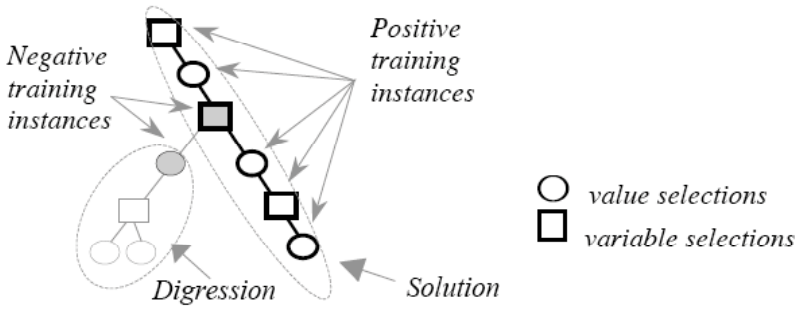

Figure 1.5: The extraction of positive and negative training instances from the trace of a successful CSP search.

guarantee that some other solution could not be found much faster, if even a single decision were different. A particular Advisor may be incorrect on some decision that resulted in a large digression, and still be correct on many other decisions in the same problem. Moreover, without supervision, we must declare what constitutes correct decisions. Clearly, the value selections that lead to unsolvable subproblems (*digressions*) are incorrect decisions. Given correct value selections, any variable ordering can produce a backtrack-free solution; a variable selection is deemed incorrect if the subsequent value assignment to that variable failed.

Given a class of problems, ACE's goal is to select Advisors and learn weights for them so that the decisions supported by the largest weighted combination of strengths lead to effective search. ACE uses a weight-learning algorithm to update its *weight profile,* the set of weights for its tier-3 Advisors. As in Figure 1.5, the learner gleans training instances from its own (likely imperfect) successful searches, and uses them to refine its search algorithm before it continues on to the next problem. *Positive training*

ACE's Weignt Learning (C, $\mathcal{A}_{var}$, $\mathcal{A}_{val}$)

```
Initialize all weights to 0.05
Until termination of the learning phase
        Identify learning problem p in C
        Search (p, A_var, A_val)
        If p is solved
                for each training instance t from p
                for each Advisor A that supported t
                        when t is a positive training instance
                                increase w(A)                    *reward*
                        when t is a negative training instance
                                decrease w(A)                    *penalize*
```

Figure 1.6: Learning on a class **C** of problems with random subsets of variable-ordering Advisors from $\mathcal{A}_{var}$ and value-ordering Advisors from $\mathcal{A}_{val}$. The *Search* algorithm is defined in Figure 1.4.

instances are those made along an error-free path extracted from a solution trace. *Negative training instances* are value selections that lead to digressions, as well as variable selections whose subsequent value assignment fails. Decisions made within a digression are not considered for learning. Figure 1.6 is a high-level description of ACE's weight-learning algorithm.

The *DWL* (Digression-based Weight Learning) algorithm reinforces Advisors' weights based on the size of the search tree and the size of each digression (Epstein, Freuder and Wallace, 2005). Size is measured by the number of *nodes* (assignments of a value to a variable). An Advisor that supports a positive training instance is given credit that depends upon the size of the search tree, relative to the minimal size of the search tree in all previous problems. An Advisor that supports a negative training instance is given penalty in proportion to the size of the resultant digression.

Small search trees indicate a good variable order, so the variable-ordering Advisors that support positive training instances from a successful small tree are highly rewarded. For value ordering, however, small search trees are interpreted as an indication that the problem was relatively easy (i.e., any value selection would likely have led to a solution), and therefore result in only small weight increments. In contrast, a successful but large search tree suggests that a problem was relatively difficult, so those value-ordering Advisors that support positive training instances in it receive substantial weight increments (Epstein, Freuder and Wallace, 2005). The weight of each Advisor is the average of all the credits and penalties it accumulated during learning.

1.5.3. Features of ACE

In ACE, the user has substantial control in the design of the search process. The user selects a propagation method (FC or MAC) and a branching method (2-way or k-way branching). The user can limit resources for solving a problem in several ways: as elapsed time allowed to solve a problem, as the number of decisions allowed during search (*steps*), or as the number of nodes. The user specifies which Advisors to use, how many problems to learn on, how many to test on, how to resolve ties (lexically or by random choice), and when, how often, and in what manner to restart search on an individual problem.

ACE also has a variety of learning mechanisms. Since weight adjustments decrease with time, ACE can monitor the weights of its tier-3 Advisors, and determine on its own to stop learning when they are *stable* (i.e., the standard deviation of the heuristics' weights over the most recent problems is small enough). ACE can transfer the results of its learning on a simple problem class to a more difficult one (*transfer learning*) or start learning on an easier class and then continue on a harder one (*bootstrapping*). It can combine its original tier-3 Advisors to learn new ones, which can then be exported to traditional solvers. On a class of graph-coloring problems, its learned weights rediscovered the well-known Brélaz heuristic (Epstein and Freuder, 2001). ACE can develop different search mechanisms for different search *stages* (search tree depths). The user can prespecify stages, but ACE can also learn the location of a break for the final stage.

ACE can also restructure itself. If there is a tier-3 Advisor whose high weight indicates that it is almost always correct for given class, ACE can *promote* that Advisor

to tier 1. ACE can also *prioritize* its tier-3 Advisors, that is, partition them into a hierarchy of subsets. If tier 3 is partitioned, only tier-3 Advisors from the top-ranked subset are initially consulted to make a decision. In the event of a tie, the next subset is consulted on only the tied actions.

1.6. Challenges in learning

Given a class of binary, solvable problems, ACE's goal is to select Advisors and learn weights for them, so that the decisions supported by the largest weighted combination of strengths lead to effective search. Our learning scenario specifies that the learner seeks only one solution to one problem at a time, and learns only from problems that it solves. There is no information about whether a single different decision might have produced a far smaller search tree. This is therefore a form of incremental, self-supervised reinforcement learning based only on limited search experience and incomplete information. Moreover, a particular heuristic may be a good choice for some decisions but a poor choice for many others in the same problem. This thesis addresses several problems ACE faces and proposes solutions to them.

If one begins with a large initial list of heuristics that contains minimizing and maximizing versions of many metrics, many of them perform poorly on a particular class of problems (*class-inappropriate heuristics*) while others perform well (*class-appropriate heuristics)*. On challenging problems, class-inappropriate heuristics occasionally acquire high weights on an initial problem, and then control subsequent decisions, so that either the problems go unsolved or the class-inappropriate heuristics receive additional rewards. The current learning mechanism for recovery is potentially

very expensive: it requires problems to be solved when class-inappropriate heuristics dominate decision making, which typically requires a very large number of nodes. The full-restart mechanism presented in Chapter 2 recognizes when its current learning attempt is not promising, abandons the responsible training problems, and restarts the entire learning process.

Given an initial set of equally weighted heuristics, many of which are class-inappropriate, it may be difficult to solve a problem within a given node limit. Because only solved problems provide training instances for weight learning, no learning can take place until some problem is solved. *Learning with random subsets*, also presented in Chapter 2, typically solves the first problem sooner, particularly when the node limit is low.

DWL, ACE's original weight-learning algorithm, gives rewards and penalties only to Advisors whose highest-strength choice matched the decision actually made. Such an approach does not reinforce other Advisors, whose highest-strength choice was not selected, even though their comments participated in voting and contributed to that decision. DWL ignores how well the Advisor discriminated among the available choices — it gives the same reinforcement to an Advisor that clearly preferred one choice above all others as to an Advisor whose comments awarded many choices the same highest strength. DWL also ignores how difficult a choice was for an Advisor, measured by the number of available choices or the constrainedness of the current subproblem. The new weight-learning algorithm proposed in Chapter 3 considers those features in the learning process.

In ACE's current decision process, each Advisor's comparative opinions (*scores*

returned by Advisor's metric) are scaled into a common range by ranking. Mere ranking, however, reflects only the preferences of one choice over another, but ignores the degree of their difference and their distribution. Chapter 3 also proposes methods for expressing Advisors' preferences. Chapter 4 discusses the significance and impact of this work and outlines plans for future work.

2. Chapter 2

This chapter begins with the experimental design used throughout this work. It describes the problem classes, parameters, and Advisors used in these experiments. The second section details the performance of some individual heuristics and motivates learning a mixture of heuristics selected from a large initial set. The third section, on full restart, describes one difficulty of learning without correct supervision and under limited resources, proposes a method that improves learning performance and demonstrates experimentally the success of the proposed method. The final section, on random subsets, addresses another difficulty of learning under limited resources, and proposes and discusses a method that improves learning performance.

2.1. Experimental design

Experiments here are performed on randomly generated classes of solvable binary CSPs:

- Model B <50, 10, 0.38, 0.2> is at the phase transition and contains very hard problems.
- Model B <50, 10, 0.18, 0.37> has the same *problem size* (number of variables and domain size), but contains somewhat easier problems.
- Model B <20, 30, 0.444, 0.5> has fewer variables but larger domains.
- Model B <30, 8, 0.26, 0.34> problems are smaller and easier.

Model B classes are henceforth identified only by their parameters <*n, m, d, t*>.

- Geometric <50, 10, 0.4, 0.82> problems are subsequently referred to as *Geo-82*.

- Composed <22, 6, 0.6, 0.1> 1 <8, 6, 0.72, 0.45> 0.115 0.05 problems have 30 variables, 22 in the central component and 8 in one satellite. They are subsequently referred to as *Comp*.

For ACE, a *learning phase* is a sequence of problems that it attempts to solve and from which it learns Advisors weights. A *testing phase* in ACE is a sequence of fresh problems to be solved with learning turned off. A *run* in ACE is a learning phase followed by a testing phase. For each experiment, results are averaged over 10 runs.

Resources for solving a problem are restricted here to a *node limit* (the number of nodes searched). In Section 2.3, the node limit for learning is an experimental parameter, and 10,000 nodes were used for testing. In all subsequent experiments, the node limits for each class are the same during both learning and testing. These limits appear in Table 2.1). When problems go unsolved under a given node limit, reported are both the number of unsolved problems and the average size of the search tree.

In all these experiments, learning is on at least 30 problems. With different methods, more learning problems may be used, but the upper bound for the total number of problems in a learning phase is always 80. During a testing phase, ACE uses only those Advisors whose weight exceeds that of their respective benchmarks. For each

Table 2.1: The node limits for problems classes in these experiments. These are used for both learning and testing phases (except in Section 2.3).

Class	Node limit
<30, 8, 0.26, 0.34>	500
Geo-82	5,000
Comp	5,000
<50, 10, 0.18, 0.37>	10,000
<20, 30, 0.444, 0.5>	20,000
<50, 10, 0.38, 0.2>	50,000

problem class, every testing phase used the same 50 problems. (The same problems were also used for testing individual heuristics in Tables 2.2 and 2.3.) When any 10 of the 50 testing problems went unsolved within the node limit, learning in that run was declared *inadequate* and further testing was halted.

Three tier-1 Advisors and 40 tier-3 Advisors are used in these experiments: 28 for variable ordering (14 pairs of heuristics and their duals) and 12 for value ordering (6 pairs of heuristics and their duals). No tier-2 Advisors were consulted. All these Advisors are described in Appendix A.

2.2. Why learning is necessary

The choice for a particular problem class of appropriate heuristics from the many touted in the constraint literature is non-trivial. The non-uniform performance of individual heuristics, as measured by average number of nodes, is illustrated in Table 2.2. Even well-trusted individual heuristics vary in their performance on different classes. For example, *Min-domain/dynamic-degree* and *Min-domain/static-degree* have similar performance on *Geo-82*, but *Min-domain/dynamic-degree* is significantly better than *Min-domain/static-degree* on <50, 10, 0.18, 0.37>.

Table 2.2: Individual heuristic search performance on three classes of problems.

Advisors	Geo-82		<50, 10, 0.18, 0.37>		<20, 30, 0.444, 0.5>	
	Nodes	Solved	Nodes	Solved	Nodes	Solved
Min-domain/dynamic-degree	258.1	98%	1365.3	100%	3403.4	100%
Min-dynamic-domain/weighted-degree	246.4	100%	1251.7	100%	3534.3	100%
Min-domain/static-degree	254.6	98%	1670.4	98%	3561.3	100%
Max-static-degree	397.7	98%	1899.1	98%	4742.1	96%
Max-weighted-degree	343.3	98%	1680.6	100%	5827.9	98%

Table 2.3 shows how Advisors used in this work fare on two classes of problems: Geo-82 and composed. A heuristic's performance is declared *inadequate* if it fails (does not find a solution within the given node limit) on at least 10 problems. Typically, on each class one of a pair of dual Advisors has much better performance than the other, but

Table 2.3: Number of nodes expanded by 28 variable-ordering heuristics on 2 classes of problems. Advisors in bold are inadequate on one class but failed on less than 10 problems on the other.

Advisors	*Geo-82*	*Comp*
Max-static-degree	432.64	inadequate
Min-static-degree	inadequate	33.15
Max-domain	inadequate	1168.71
Min-domain	705.22	373.22
Min-FF2.2	inadequate	inadequate
Max-FF2.2	546.98	31.59
Max-backward-degree	721.50	inadequate
Min-backward-degree	inadequate	137.78
Max-dynamic-degree	456.20	876.51
Min-dynamic-degree	inadequate	32.84
Min-value-pairs	inadequate	32.05
Max-value-pairs	inadequate	1068.94
Min-static-connected-edges	inadequate	32.90
Max-static-connected-edges	422.07	inadequate
Max-static-less-connected-edges	inadequate	32.73
Min-static-less-connected-edges	287.25	inadequate
Max-dynamic-connected-edges	479.93	1049.94
Min-dynamic-connected-edges	inadequate	32.48
Max-dynamic-less-connected-edges	inadequate	32.62
Min-dynamic-less-connected-edges	321.93	772.53
Min-domain/static-degree	284.70	inadequate
Max-domain/static-degree	inadequate	334.80
Min-domain/dynamic-degree	278.09	inadequate
Max-domain/dynamic-degree	inadequate	532.22
Min-weighted-degree	inadequate	32.83
Max-weighted-degree	383.26	50.44
Min-dynamic-domain/weighted-degree	238.53	57.67
Max-dynamic-domain/weighted-degree	inadequate	532.18

there are cases where both are successful on one class (e.g., *Min-weighted-degree* and *Max-weighted-degree* on the composed class) or unsuccessful (e.g., *Min-value-pairs* and *Max-value-pairs* on Geo-82). Many Advisors (in bold) have inadequate performance on one class but are successful on the other. There are also Advisors that performs inadequately on both classes (e.g., *Min-FF2.2*) and Advisors that perform well on both classes (e.g., *Max-weighted-degree*).

Composed problems are particularly challenging for most commonly-used heuristics, but often successfully solved by their duals, as shown in Table 2.3. The central component in these problems was deliberately made substantially larger, *looser* (with lower tightness), and *sparser* (with lower density) than its satellite. Once a solution to the subproblem defined by the satellite is found, it is relatively easy to extend that solution to the looser and sparser central component. In contrast, if one extends a partial solution for the subproblem defined by the central component to the satellite variables, inconsistencies eventually arise deep within the search tree. Typically such problems are either solved with minimal backtracking or go unsolved after hundreds of thousands of steps.

Despite the low density of the central component in a *Comp* problem, its variables' degrees are often larger than those in the significantly smaller satellite. Some traditional heuristics (e.g., *Max-static-degree*) tend to select variables from the much larger central component first, and therefore fail to solve many such problems within a reasonable node limit. In contrast, the decidedly untraditional *Min-static-degree* heuristic tends to prefer variables from the small satellite and thereby detects inconsistencies much earlier.

The characteristics of such composed problems are often found in real-world problems (Lecoutre, Boussemart and Hemery, 2004; Otten, Grönkvist and Dubhashi, 2006; Petrie and Smith, 2003). To achieve good performance without knowledge about a problem's structure, therefore, it is advisable to consider many popular heuristics along with their duals.

2.3. Full Restart

On challenging problems, class-inappropriate Advisors occasionally acquire high weights on an initial problem, and then control subsequent decisions. (Some samples of weight profiles appear in Appendix B.) As a result, a problem needs a large number of decisions to be solved. Under unlimited resources, DWL will recover from such a situation. Class-inappropriate Advisors typically generate large search trees and large digressions, so DWL imposes large penalties and provides small credits to variable-ordering Advisors that lead decisions. With significantly reduced weights, those Advisors no longer dominate the set of class-appropriate Advisors. Solving a hard problem without good heuristics is typically computationally very expensive and may demand resources that are unavailable. If a problem goes unsolved under a given node limit, no weight changes occur and there is no learning.

Rather than wait for the weights of class-appropriate Advisors to recover from expensive solutions, *full restart* abandons the learning process (and any learned weights) and begins learning afresh on new problems (Petrovic and Epstein, 2006a). Full restart monitors the frequency and distribution of unsolved problems under a reasonably low node limit. If it deems the current learning attempt not promising, the responsible training

problems are abandoned, and the entire learning process restarts with freshly initialized weights.

The *restart strategy* here is defined with a *full restart threshold (k l)*, which performs a full restart after failure on *k* problems out of the last *l*. This seeks to avoid full restarts when multiple but sporadic failures are actually due to uneven problem difficulty rather than to an inadequate weight profile. Problems that went unsolved under initial weights, before any learning occurred (*early failures*) are not counted toward full restart. If the first 30 problems go unsolved under the initial weights, learning is terminated.

The node limit is a critical parameter for full restart. Because ACE abandons a problem if it does not find a solution within the node limit, the node limit is the criterion for unsuccessful search. Since the full restart threshold directly depends upon the number of failures, the node limit is the performance standard for full restart. The node limit also controls resources; the lengthily searches permitted by high node limits are expensive.

The experiments here on <30, 8, 0.26, 0.34>, *Geo-82* and <50, 10, 0.18, 0.37> problems with and without full restart under a (3 4) threshold illustrate benefits of full restarts. Testing phases with 10 unsolved problems were terminated and learning for such a run was considered *inadequate*. (Typically, few problems were solved at all during any inadequate run, and the 10 failures appeared within the first 15 problems.) Otherwise, learning was declared *successful*. The *learning cost* is the total number of nodes during the learning phase of a run, calculated as the product of the average number of nodes per problem and the average number of problems per run. The learner's performance here is measured by the number of successful runs (out of 10) and the learning cost across a range of node limits.

For all three classes, under every node limit tested, full restart produced at least as many successful runs as did experiments without full restart. At lower node limits, better testing performance is obtained with a learning cost similar to or slightly higher than the cost without full restart. At higher node limits, learning cost is considerably lower with full restart.

Table 2.4 and Figures 2.1 and 2.2 show the impact of full restart on <30, 8, 0.26, 0.34> problems. With very low node limits (200 and 300 nodes), neither

Table 2.4: Learning and testing performance without full restart and with full restart after 3 out of 4 problems from <30, 8, 0.26, 0.34> go unsolved.

	Without full restart				
	Learning				Testing
Node limit	Number of learning problems	Number of nodes per problem	Learning cost	Number of full restarts	Number of successful runs
200	30	152.2	4,565.9	0	6
300	30	168.1	5,043.7	0	8
400	30	232.7	6,982.3	0	7
500	30	269.8	8,094.1	0	7
600	30	325.9	9,777.7	0	6
700	30	399.0	11,971.4	0	6
800	30	351.4	10,541.9	0	7
5000	30	952.8	28,582.7	0	9
	With full restart				
	Learning				Testing
Node limit	Number of learning problems	Number of nodes per problem	Learning cost	Number of full restarts	Number of successful runs
200	36.0	147.6	5,313.7	0.7	8
300	38.7	189.9	7,349.3	1.0	9
400	36.3	187.5	6,805.6	0.5	10
500	37.8	206.4	7,800.2	0.6	10
600	31.9	234.5	7,480.9	0.3	9
700	31.9	256.1	8,168.9	0.3	9
800	31.9	278.7	8,891.3	0.3	9
5000	31.7	405.1	12,842.0	0.3	10

method was able to solve all the problems. During learning, many problems went unsolved under a low node limit and therefore did not provide training instances. Moreover, some inadequate runs were the result of consecutive failure on the first 30 problems, so learning was terminated with initial weights under both methods.

When the node limit was somewhat higher (400 and 500 nodes), more problems were solved, more training instances were available and more runs were successful.

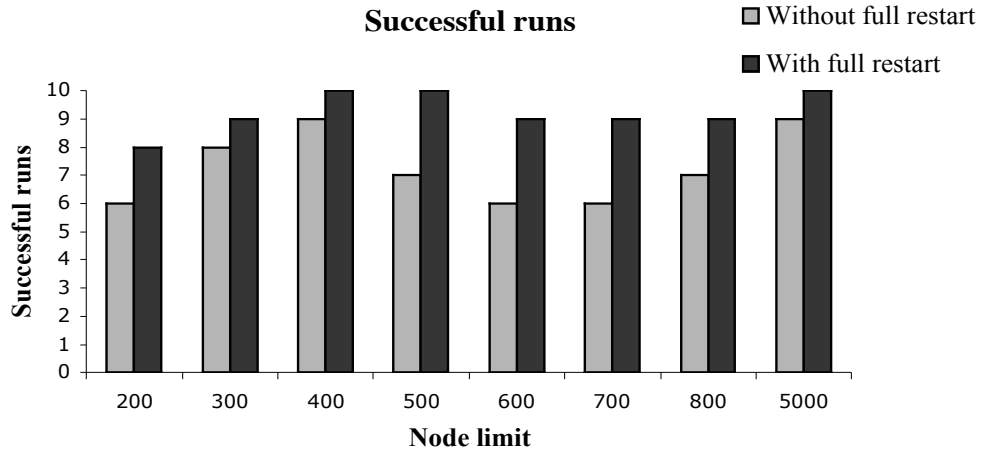

Figure 2.1: Number of successful runs (out of 10) on <30, 8, 0.26, 0.34> problems, under different node limits.

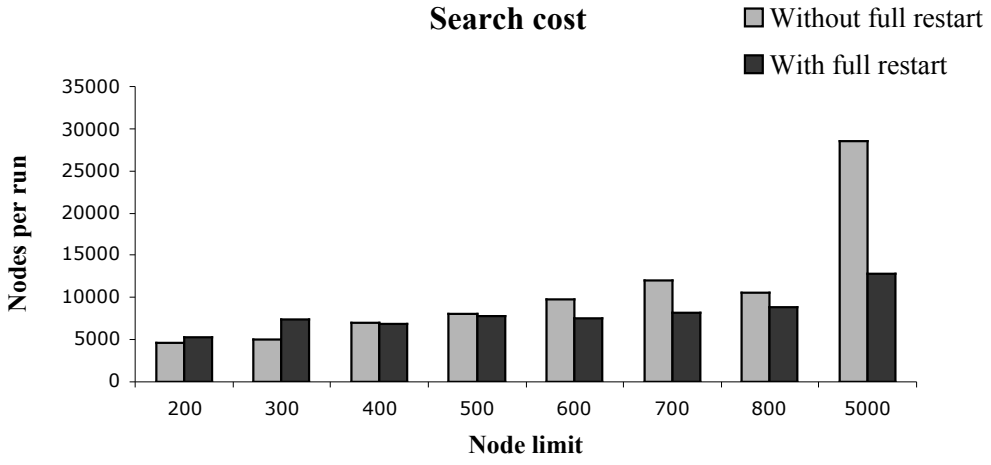

Figure 2.2: Learning cost, measured by the average number of nodes per run, on <30, 8, 0.26, 0.34> problems, under different node limits.

These reasonably low node limits set a high standard for the learner: only weight profiles well-tuned to the class will solve problems within them and thereby provide good training instances. Further increases in the node limit (600, 700 and 800 nodes), however, do not further increase the number of successful runs. Under larger node limits, problems were solved even with weight profiles that are not particularly good for the class, and

Table 2.5: Learning and testing performance with full restart after 3 out of 4 problems from *Geo-82* go unsolved.

	Without full restart		**With full restart**			
Node limit	Successful runs	Learning cost	Number of learning problems	Full restarts	Successful runs	Learning cost
5,000	4	71,329.8	59.4	3.4	10	127,344.0
10,000	4	128,392.1	45.9	1.8	10	140,325.1
50,000	6	455,352.5	43.9	1.8	9	665,507.7
100,000	4	980,669.1	37.4	0.6	9	649,551.0

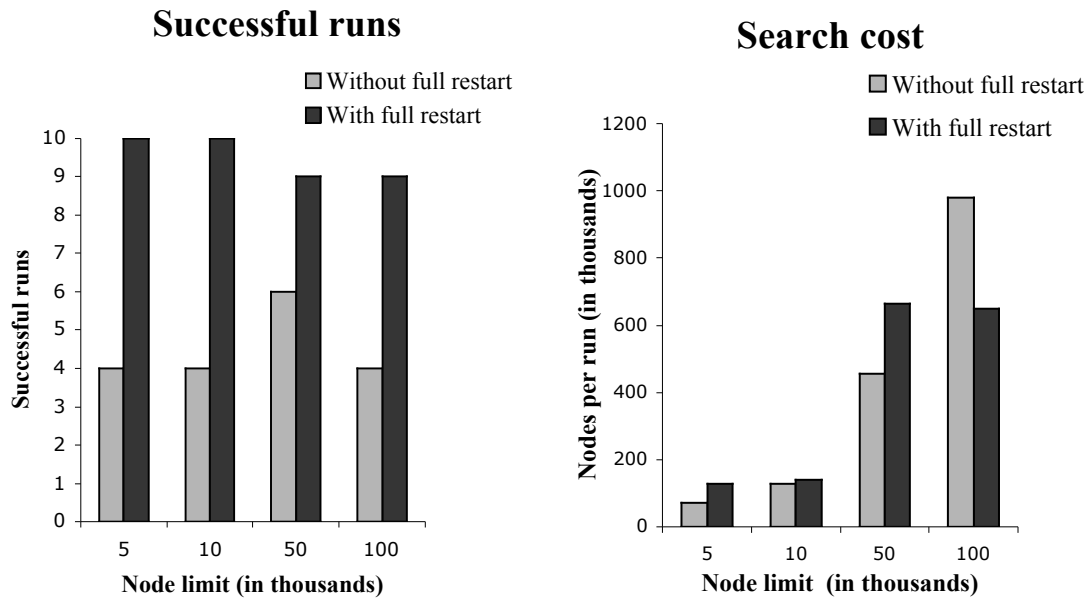

Figure 2.3: Learning and testing performance with full restart on Geo-82 problems.

may have produced training instances that were not appropriate. Under extremely high node limits (5000 nodes), problems were solved even under an inadequate weight profiles, but the weight-learning mechanism was able to recover a good weight profile, and again the number of successful runs increased.

Table 2.5 and Figure 2.3 also show more successful with full restart and reduced learning cost under higher node limits for Geo-82 problems. On the more difficult <50, 10, 0.18, 0.37> problems, Table 2.6 and Figure 2.4 show better testing performance

Table 2.6: Learning and testing performance with full restart after 3 out of 4 problems from <50, 10, 0.18, 0.37> go unsolved.

	Without full restart		**With full restart**			
Node limit	Number of successful runs	Learning cost	Number of learning problems	Number of full restarts	Number of successful runs	Learning cost
10,000	4	269,836.7	33.1	0.2	4	300,808.7
50,000	4	1,102,694.7	42.6	1.0	7	1,566,292.8
100,000	3	2,382,896.1	41.0	1.0	6	2,613,164.6

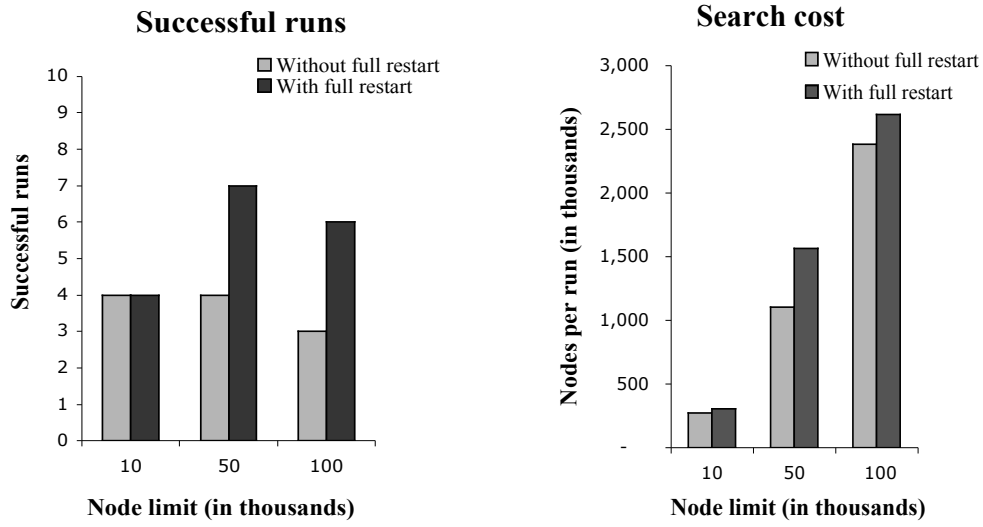

Figure 2.4: Testing and learning performance on <50, 10, 0.18, 0.37> problems.

under full restart. Under a 10,000 node limit, all six inadequate runs entered the testing phase after 30 learning problems went unsolved and therefore without any weight learning. This resulted in few learning problems and few full restarts. Given the difficulty of these problems, it is infeasible to demonstrate the expected behavior of learning cost for higher node limits.

Full restart is not a panacea. On *Comp* problems, for example, with many heuristics, some problems go unsolved even under a very high node limit, and under a very low node limit, many problems are solved. Because failures are sporadic, they do not trigger full restart. The use of full restart does not improve learning, but it does not harm it either.

In conclusion, with higher node limits, weights can eventually recover without the use of full restart, but recovery is more expensive. With lower node limits, the cost of learning with full restart is slightly higher than without it. The learner fails on all the difficult problems, and even on some of medium difficulty, repeatedly triggering full restart until the weight profile is good enough to solve almost all the problems. Full restart abandons some problems and uses additional problems, which increases the cost of learning. The difference in cost is small, however, since each problem's cost is limited by the relatively low node limit. As the node limit increases, full restart is able to avoid inadequate runs, but at a higher cost. It takes longer to trigger full restart because the learned weight profile is good enough, so that failures are less frequent. Moreover, with a high node limit, every failure is expensive. When full restart eventually triggers, the prospect of relatively extensive effort on further problems is gone. By detecting and

eliminating unpromising learning runs early, full restart avoids many costly searches and drastically reduces overall learning cost.

2.4. Random subsets

The interaction among heuristics can also serve as a filter during learning. Given an initial set of heuristics that is large and inconsistent, many class-inappropriate heuristics may combine to make bad choices, and thereby make it difficult to solve the first problem within a given node limit. Because only solved problems provide training instances for weight learning, no learning can take place until some problem is solved.

Rather than consult all its Advisors at once, ACE can use *learning with random subsets* to randomly select a new subset of Advisors for each problem, consult them, make decisions based on their comments, and update only their weights (Petrovic and Epstein, 2007b; Petrovic and Epstein, In press). Figure 2.5 is a high-level algorithm for learning with random subsets.

Since global search is complete, under a large enough node limit, every problem will eventually be solved, regardless of the weight mixture and the weight profile. There are, however, many factors that determine the number of steps required to find a solution.

These include:

- The number and the degree of appropriateness of the participating heuristics, as demonstrated in Section 2.2.
- The problem difficultly, which may vary substantially within a given class for a given search algorithm (Hulubei and O'Sullivan, 2005).
- The frequency with which the heuristics comment, their discriminative power, and their accuracy (Epstein, 2004b). For example, even appropriate heuristics must comment frequently if they are to direct decision making during search.

For a fixed node limit and set of heuristics, an assumption here is that the ratio of class-appropriate to class-inappropriate heuristics determines whether a problem is likely to be solved.

When class-inappropriate heuristics predominate in a random subset, the problem is unlikely to be solved and no learning occurs. The repeated selection of a random subset of heuristics for each new problem, however, will eventually produce some subset S with a majority of class-appropriate heuristics that solves its problem within a reasonable resource limit. As a result, all participating Advisors in S will have their weights adjusted. On the next problem, the new random subset S' is likely to contain some low-weight Advisors outside of S, and some reselected from S. Any previously-successful Advisors from S that are selected for S' will have larger positive weights than the other Advisors in

LearnFromRandomSubsets ($\mathbf{C}$, $\mathcal{A}_{var}$, $\mathcal{A}_{val}$)

```
Initialize all weights to 0.05
Until termination of the learning phase
        Identify a learning problem p in C
        Generate or accept x and y in [0,100]
        Randomly select a subset S_var of x percent of the variable Advisors from A_var
        Randomly select a subset S_val of y percent of the value Advisors from A_val
        Search (p, S_var ∪ S_val)
        If p is solved
                for each training instance t from p
                for each Advisor A that supported t
                        when t is a positive training instance
                                increase w(A)                    *reward*
                        when t is a negative training instance
                                decrease w(A)                    *penalize*
        else when the full restart criteria are satisfied
        initialize all weights to 0.05
```

Figure 2.5: Learning on a class $\mathbf{C}$ of problems with random subsets of variable-ordering Advisors from $\mathcal{A}_{var}$ and value-ordering Advisors from $\mathcal{A}_{val}$. The *Search* algorithm is defined in Figure 1.4.

S', and will therefore heavily influence search decisions. If *S* succeeded because it contained more class-appropriate than class-inappropriate heuristics, $S \cap S'$ is also likely to have more class-appropriate heuristics and thereby solve the new problem, so again those that participate in correct decisions will be rewarded. On the other hand, in the less likely case that the majority of $S \cap S'$ consists of reinforced, class-inappropriate heuristics, the problem will likely go unsolved, and the class-inappropriate heuristics will not be rewarded further.

Figure 2.6 illustrates how the weights of seven heuristics converged during learning with random subsets and some Advisors recovered from inadequate weights. Here the problems were drawn from <50, 10, 0.38, 0.2>, and 30% of the Advisors were randomly selected for each problem. Plateaus in weights correspond to problems where the particular heuristic was not selected for the current random subset, or the problem was not solved so no learning and weight change occurred. The first four problems were early failures. The fifth problem was solved, but class-inappropriate Advisors received high weights. On the next several problems, when highly-weighted but inadequate

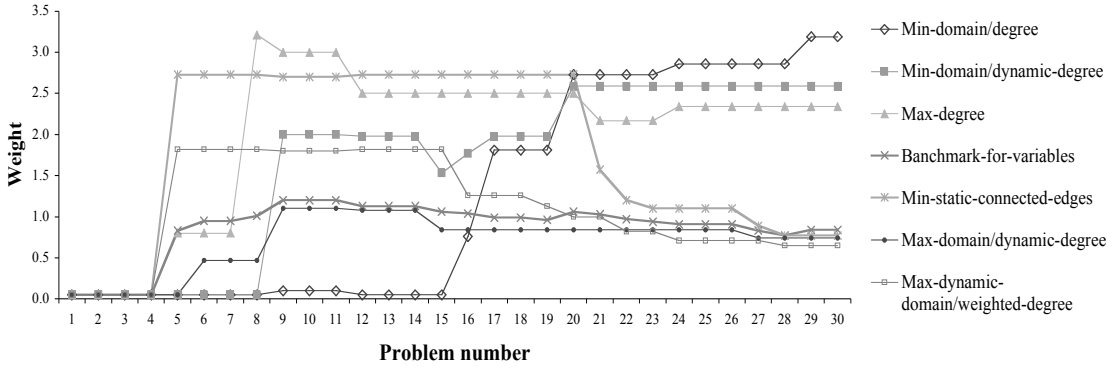

Figure 2.6: Weights of seven Advisors during learning after each of 30 problems in a single run.

heuristics were reselected, the problem was not solved, and no weight changes occurred, but on solved problems, some class-appropriate Advisors gained high weights and began to dominate decisions, so that disagreeing class-inappropriate Advisors had their weights reduced. After the 21st problem, when the weight of *Min-static-connected-edges* significantly decreased, the weights clearly separate the class-appropriate Advisors from the class-inappropriate ones. Then, as learning progressed, the weights stabilized.

In experiments that illustrate the benefits of random subsets, three different ways to choose the Advisors to each problem were implemented:

- Use all the Advisors on every problem.
- Choose a fixed percentage q of the variable-ordering Advisors and q of the value-ordering Advisors, without replacement. Testing was performed for both $q = 30\%$ and $q = 70\%$.
- For each problem, select a random percentage r in [30, 70]. Choose r percent of the variable-ordering Advisors and r percent of the value-ordering Advisors, without replacement.

Table 2.7 compares learning with random subsets of Advisors to learning with all the Advisors at once, on problems in <50, 10, 0.18, 0.37> under different node limits. With all the Advisors, some learning phases were terminated after 30 unsolved problems, hence no learning occurred and some runs were inadequate. With random subsets, however, adequate weights were learned in every run. The number of early failures, the number of full restarts, and the number of learning problems also decreased with random subsets.

When random subsets are used with a relatively low node limit on hard problems, failure to solve a problem does not necessarily indicate that the currently assigned weights are inadequate. Particularly when random subsets are small, selections of Advisors for different problems may not overlap, and highly-weighted class-appropriate heuristics may not have any influence on some subsequent problems. Full restart triggered by a small number of failures after one problem is solved may frequently abandon correctly learned weights and thereby reduce learning efficiency. In the following experiments on <50, 10, 0.18, 0.37>, <20, 30, 0.444, 0.5> and <50, 10, 0.38, 0.2> problems, the full restart threshold was failure on 5 out of the most recent 7 problems.

Table 2.7: Improved learning performance with random subsets on problems from <50, 10, 0.18, 0.37>.

All Advisors				
Node limit	Number of learning problems	Number of early failures	Number of full restarts	Number of runs without any learning
10000	33.1	28.7	0.2	6
50000	42.6	27.4	1.0	3
100000	41.0	21.8	1.0	3
Random size subsets of Advisors				
Node limit	Number of learning problems	Number of early failures	Number of full restarts	Number of runs without any learning
10000	31.8	2.9	0.3	0
50000	31.4	2.1	0.2	0
100000	30.8	3.8	0.1	0

Table 2.8: Learning with different methods of selection for the participating Advisors subsets. (*) indicates that only 9 runs were completed under overall experimental resources.

	Learning			**Testing**		
Advisors in learning	Number of learning problems	Number of unsolved problems	Number of early failures	Number of successful runs	Number of nodes	Percentage solved
Comp						
All	30.0	2.5	0	10	298.45	95.40%
Random 70%	31.4	1.4	0	10	184.6	97.60%
Random 30%-70%	30.0	0.8	0	10	205.68	96.80%
Random 30%	30.0	1.3	0	10	238.09	96.40%
<30, 8, 0.26, 0.34>						
All	34.1	6.3	5.0	10	102.98	100.00%
Random 70%	33.4	4.0	2.2	10	102.31	100.00%
Random 30%-70%	30.0	1.8	0.9	10	102.03	100.00%
Random 30%	38.7	8.9	2.8	10	103.42	100.00%
Geo-82						
All	44.8	13.7	7.1	10	192.73	98.60%
Random 70%	35.1	5.1	2.1	10	192.45	98.60%
Random 30%-70%	35.5	6.2	2.4	10	194.35	98.40%
Random 30%	36.1	6.4	0.9	10	195.93	98.40%
<50, 10, 0.18, 0.37>						
All	32.2	27.8	27.0	4	–	–
Random 70%	30.0	5.3	5.2	10	1,239.10	100.00%
Random 30%-70%	31.2	4.8	1.9	10	1,379.68	100.00%
Random 30%	31.2	6.4	3.1	10	1,317.51	99.80%
<20, 30, 0.444, 0.5>						
All	41.1	10.9	5.2	10	3,424.48	100.00%
Random 70%	32.0	3.2	1.0	10	3,323.81	100.00%
Random 30%-70%	32.6	5.3	2.5	10	3,517.91	100.00%
Random 30%	33.8	6.8	1.0	10	2,678.56	100.00%
<50, 10, 0.38, 0.2>						
All (*)	32.9	21.2	21.2	5	–	–
Random 70%	36.7	15.5	9.8	10	13,767.79	90.60%
Random 30%-70%	32.2	12.0	7.2	9	–	–
Random 30%	36.8	14.1	4.4	10	13,708.58	91.80%

On *Comp* problems, early failures are not an issue, even with a very low node limit, so there is no need for random selection of Advisors. Nevertheless, Table 2.8 shows that learning with random subsets does not harm performance on a problem class where it is unnecessary. On the other classes in Table 2.8, learning performance improved with random subsets under low node limits: there were no inadequate runs, and the number of unsolved problems and the number of early failures were reduced.

Table 2.9 demonstrates that using random subsets significantly reduces learning time. The time to select a variable or a value is not necessarily directly proportional to the number of selected Advisors. This is primarily because the pair of Advisors that minimize and maximize the same metric share the same fundamental computational cost: calculating their common metric. For example, the bulk of the work for *Min-product-domain-value* lies in the one-step lookahead that calculates the products of the domain sizes of the neighbors after each potential value assignment. Consulting only *Min-product-domain-value* and not *Max-product-domain-value* will therefore not significantly reduce computational time. Moreover, the metrics for some Advisors are based upon metrics already calculated for others. For example, the two Advisors whose metric is the ratio of dynamic domain size to weighted degree are relatively fast if those two metrics were calculated earlier by other Advisors for their own use. This is why, for example, in Table 2.9 with <50, 10, 0.18, 0.37> problems, removing 70% of the Advisors saved only 44.61% of the computation time for each decision. The reduction in total computation time is even more dramatic, because it incorporates the number of learning problems and nodes per problem.

Table 2.9 Percentage of computation time during learning with random subsets, compared to computation time with all Advisors.

Advisors in learning	Learning	
	Computation time per decision	Learning time per run
Comp		
All	100.00%	100.00%
Random 70%	92.43%	53.08%
Random 30%-70%	61.25%	23.91%
Random 30%	38.09%	24.41%
<30, 8, 0.26, 0.34>		
All	100.00%	100.00%
Random 70%	75.04%	64.92%
Random 30%-70%	61.61%	45.57%
Random 30%	51.11%	70.32%
Geo-82		
All	100.00%	100.00%
Random 70%	87.65%	38.42%
Random 30%-70%	74.01%	37.95%
Random 30%	65.52%	35.94%
<50, 10, 0.18, 0.37>		
All	100.00%	100.00%
Random 70%	74.30%	24.39%
Random 30%-70%	65.84%	20.74%
Random 30%	55.39%	21.85%
<20, 30, 0.444, 0.5>		
All	100.00%	100.00%
Random 70%	84.45%	36.12%
Random 30%-70%	76.52%	39.33%
Random 30%	80.59%	32.56%
<50, 10, 0.38, 0.2>		
All	100.00%	100.00%
Random 70%	78.22%	64.42%
Random 30%-70%	67.46%	45.81%
Random 30%	60.74%	51.07%

The robustness of learning with random subsets is demonstrated with experiments that begin with fewer Advisors, a majority of which are class-inappropriate. Based on weights from successful runs in Table 2.8, Advisors were first identified as class-appropriate or class-inappropriate for <50, 10, 0.18, 0.37> problems. ACE was then provided with three different sets of variable Advisors A_{var} in which class-inappropriate Advisors outnumbered class-appropriate ones. (The full list of Advisors in each set appears in Appendix C.) Set-1 consisted of 6 class-appropriate Advisors and 9 class-inappropriate Advisors. To create Set-2, two Advisors that typically earn high weights and lead decisions in a mixture (*Min-domain/dynamic-degree* and *Max-weighted-degree*) were removed and replaced with more "neutral" class-appropriate Advisors (*Max-backward-degree and Max-dynamic-degree),* whose weight are typically lower in successful mixtures. Set-3 was biased even further toward class-inappropriate Advisors: *Max-backward-degree* and *Max-dynamic-degree* were removed, leaving Set-3 with only 4 class-appropriate and 9 class-inappropriate Advisors. Full restart was performed when 5 out of 7 problems went unsolved.

As demonstrated in Table 2.10, when all the provided Advisors were consulted, the predominance of class-inappropriate Advisors effectively prevented the solution of any problem under the given node limit and no learning took place. When learning with random subsets, as the size of the random subsets decreased, the number of successful runs increased. There were also significantly fewer unsolved problems; in particular, there were fewer early failures.

More early failures occur and more learning problems are required when random subsets are relatively large (70% vs. 30%). Furthermore, all the inadequate runs with

subset size 70% came after each of the first 30 problems went unsolved and testing relied on initial weights. Intuitively, if there are few class-appropriate heuristics in A, the probability that they are selected as a majority in a larger subset is small (0 if the subset size is more than twice the number of class-appropriate Advisors). For example, given a class-appropriate Advisors, and b class-inappropriate Advisors, the probability that the majority of r randomly-selected Advisors is class-appropriate is

$$p = \sum_{k=\left\lfloor \frac{r}{2}+1 \right\rfloor}^{r} \frac{\binom{a}{k}\binom{b}{r-k}}{\binom{a+b}{r}} \qquad [2.1]$$

and the expected number of trials until the subset has a majority of class-appropriate Advisors (the mean of this geometric distribution) is

$$\sum_{i=1}^{\infty} i(1-p)^{i-1} p = \frac{1}{p} \qquad [2.2]$$

When there are more class-inappropriate Advisors ($a < b$), a smaller set is more likely to have a majority of class-appropriate Advisors. For example, in Set-1 and Set-2, where $a = 6$ and $b = 9$, if $r = 4$, [2.1] evaluates to 0.14 and [2.2] to 7. For $a = 6$, $b = 9$, and $r = 10$, however, the probability of a randomly selected subset with a majority of class-appropriate heuristics is only 0.042 and the expected number of trials until the subset has a majority of class-appropriate Advisors is 23.8.

The number of early failures naturally depends also on the node limit and on how appropriate Advisors are for the given class. For example, the number of early failures under Set-1 is smaller than under Set-2 since two Advisors that are particularly good for given class are removed. Even under small subsets, a combination of mediocre Advisors

did not manage to succeed in one run. Most of the early failures occurred in Set-3, where the number of inadequate Advisors was much higher than the number of adequate Advisors.

Weights converge faster when subsets are larger. With random subsets of 70% in Table 2.10, there is a particularly high percentage of early failures out of all failures. Thus, once some problem is solved, not many additional failures occur. When the random subsets are smaller, subsequent random subsets are less likely to overlap with those that preceded them, and therefore less likely to include Advisors whose weights have been

Table 2.10 Learning with more class-inappropriate than class-appropriate Advisors on problems in <50, 10, 0.18, 0.37>.

	Learning				**Testing**
Advisors in learning	Number of learning problems	Percentage of unsolved problems	Percentage of early failures out of all problems	Percentage of early failures out of all failures	Number of successful runs
Set 1: 6 class-appropriate, 9 class-inappropriate Advisors					
All	30.0	100.00%	100.00%	100.00%	0
Random 70%	34.8	67.00%	60.90%	91.00%	7
Random 30%-70%	32.3	33.70%	26.00%	77.10%	10
Random 30%	33.4	37.10%	15.30%	41.10%	10
Set 2: 6 class-appropriate, 9 class-inappropriate Advisors (stronger polarization)					
All	30.0	100.00%	100.00%	100.00%	0
Random 70%	32.1	79.40%	75.40%	94.90%	6
Random 30%-70%	33.2	44.30%	31.90%	72.10%	8
Random 30%	40.6	52.00%	16.00%	30.50%	9
Set 3: 4 class-appropriate, 9 class-inappropriate Advisors					
All	30.0	100.00%	100.00%	100.00%	0
Random 70%	30.0	100.00%	100.00%	100.00%	0
Random 30%-70%	38.7	66.10%	45.50%	68.80%	6
Random 30%	41.8	59.00%	32.00%	53.60%	10

revised. As a result, failures occur often, even after some class-appropriate heuristics receive high weights. When the subset size varies, however, smaller subsets allow better learning when class-appropriate Advisors are outnumbered, while a larger size make overlap more likely, and thereby speeds learning. When there are roughly as many class-appropriate as class-inappropriate Advisors (as in Table 2.8), the subset sizes are less important.

3. Chapter 3

When each heuristic reflects an underlying metric, nuances from comparative opinions can be exploited to improve search and learning performance. A variety of approaches here consider both the scores returned by the metrics on which these heuristics rely and the distributions of those scores across a set of possible actions. *RSWL* (Relative Support Weight Learning) is a new weight-learning algorithm that considers additional factors to make decisions and to update weights during learning. Section 3.1 considers both the distribution of a heuristic's degree of support for a choice and the difficulty of the subproblem at the time the choice is made. Section 3.2 considers the distribution of each heuristic's degree of support in voting to select variables and values while solving a problem. Section 3.3 describes experimental results that support the effectiveness of those methods.

3.1. Relative support weight learning

3.1.1. Relative support

In ACE, an Advisor does not only select one variable or value – it comments on multiple choices. Recall that each heuristic has its own underlying descriptive metric, a function from the set of available choices to the real numbers. That metric returns a *score* for each choice. A metric can return large scores, for example, the size of a potential search tree after some value assignment, estimated as the product of all the dynamic domain sizes. A metric can also return small scores, for example, the likelihood of search failure estimated as the average tightness over the neighbors in the original constraint graph to

the power of the static degree of the variable. To combine heuristics equitably, scores returned by metrics are scaled into some common range. The extent of an Advisor's support is expressed by the *strength* that each heuristic assigns to the available choices (Epstein and Freuder, 2001).

ACE uses limited information from heuristics' scores. All comments participate in selecting the *action a* (the winner of the voting in Figure 1.4). For credits and penalties, however, DWL considers only those Advisors that assign the highest strength to the action. In such a scenario, an Advisor that contributed to the selection of an action only by its second-highest strength will not get any credit; indeed it will have its weight reduced because it will have commented one more time but not have been correct.

In contrast, the new weight-learning algorithm, RSWL, considers all the strengths from an Advisor's comments, not only its highest. Given a set C of choices, for a particular $c \in C$, let $s(A, c, C)$ be the strength of Advisor A's preference for choice c. The average support of Advisor A for the choices in C, $avg(A, C)$, is the average of the strengths in all A's comments across the choices c_i in C:

$$avg(A,C) = \frac{\sum_{i=1}^{|C|} s(A,c_i,C)}{|C|} \qquad [3.1]$$

The *relative support* of Advisor A for action a, $rs(A, a)$, is the normalized difference between the strength the Advisor assigned to a and the Advisor's average support across all comments on C:

$$rs(A,a,C) = \frac{s(A,a,C) - avg(A,C)}{avg(A,C)} \qquad [3.2]$$

Under RSWL, an Advisor supports an action if its relative support for that action is positive (i.e., if $s(A, a, C) > avg(A, C)$), and opposes an action if its relative support is

negative (i.e., if $s(A, a, C) < avg(A, C)$). If $s(A, c, C)=0$, then no credit or penalty is assigned.

3.1.2. Credits and penalties in RSWL

Under DWL, credits and penalties (*weight adjustments*) are assigned according to the size of the completed search tree and the size of its digressions. DWL considers neither the degree of an Advisor's support when it determines weight adjustments nor how difficult it was to make a decision. In contrast, RSWL's credits are directly proportional to the relative support of an Advisor for an action. Support for a correct decision with near-average strength earns only a small credit, while support with substantially greater than average strength indicates a clear preference from the Advisor and receives a proportionally larger credit. Under RSWL, penalties for the support of a negative training instance are proportional to relative support, but inversely proportional to the number of choices from among which the action was selected. The penalty for an incorrect decision from among many choices is smaller than the penalty for an incorrect decision from among only a few choices.

Two variants of RSWL further emphasize the idea that the difficulty of the decision should impact weight adjustments. On an easy subproblem, credits should be reduced because most decisions lead to a solution, and because such credit increases the weights of the Advisors that led to it (whose weights were presumably already high). On easy problems, however, an incorrect decision should be rare and therefore severely penalized. This is addressed with RSWL-κ and RSWL-d.

The *RSWL-κ* algorithm determines whether a credit or a penalty is given by the proximity to 1 of the kappa value of the current subproblem. As discussed in Section

1.2.3, for every search algorithm, and for a fixed problem size, hard problem classes have κ near 1. Although κ was intended for a class, RSWL-κ uses it as a measure of subproblem difficulty throughout search. Let κ_s be the estimated difficulty of the current subproblem *s,* and let *k* be a user-specified parameter. RSWL-κ gives credit to an Advisor only when it supports a positive training instance derived from a search state where $|\kappa_s - 1| < k$. RSWL-κ penalizes an Advisor only when it supports a negative training instance and the corresponding state has $|\kappa_s - 1| > k$.

Because calculating κ_s on every training instance is computationally expensive, another variant, the *RSWL-d* algorithm, uses the number of unassigned variables at the current search node as a rough, quick estimate of problem hardness. Decisions at the top of the search tree are known to be more difficult (Ruan, Kautz and Horvitz, 2004). For a given parameter *h*, no penalty is given at all for any decision at the top *h* percent of the nodes in the search tree, and no credit is given for any decision below them.

3.2. Heuristics' preferences

Recall that Advisor *A* calculates some score on each choice *c* in the set *C* of all current choices. In this way, an Advisor's metric creates an ordered partition $\mathscr{C}_A(C) = \{C_1, C_2, \ldots, C_l\}$, where choices in the same subset C_i share a common score v_i. These subsets are ordered decreasingly by their Advisor's predilection: if *A* prefers choices from C_i to choices from C_j, then $i < j$.

In all the methods considered here for expressing heuristics' preferences, the number of choices to which an Advisor assigns non-zero values depends on a user-specified constant *p*. If there are more than *p* subsets (i.e., $p < |\mathscr{C}_A(C)|$), an Advisor

assigns non-zero strength only to the choices in the first p subsets. If $p \geq |\mathscr{C}_A(C)|$, every choice will receive a non-zero strength. Choices from C_k for $k > min(p, |\mathscr{C}_A(C)|)$ are given strength 0; this makes them irrelevant during voting. Note that across all problems and all decisions within a problem, p is constant, while the partition and it size $|\mathscr{C}_A(C)|$ changes dynamically.

The strength of A's support is calculated for each $C_i \in \mathscr{C}_A(C)$. All choices in the same subset of a partition receive the same strength, calculated by some function g that depends not only on the position of C_i in the ordered partition $\mathscr{C}_A(C)$, but also on other characteristics of the partition $\mathscr{C}_A(C)$ induced by the Advisor's metric:

$$\forall c \in C_i,\ s(A, c, C) = g(\mathscr{C}_A(C)) \qquad [3.3]$$

This section considers four alternative preference expression methods for g; each converts metric scores to some common scale. They are illustrated by Table 3.1 on the example in Figure 3.1, whose structure illustrates the reasons for these new preferences methods. There are 12 variable choices, C=$\{X, Y_1,\ldots,Y_{10}, Z\}$. The *degree* metric partitions C into 3 subsets, $\mathscr{C}_{degree} = \{\{X\}, \{Y_1,\ldots Y_{10}\}, \{Z\}\}$ with respective scores v_1=11, v_2=2 and v_3=1 (shown in the first line of Table 3.1). Note that the degree of variable X is significantly higher than the degrees of the other variables and that the size of the subset of Y variables is significantly larger than the other two subsets.

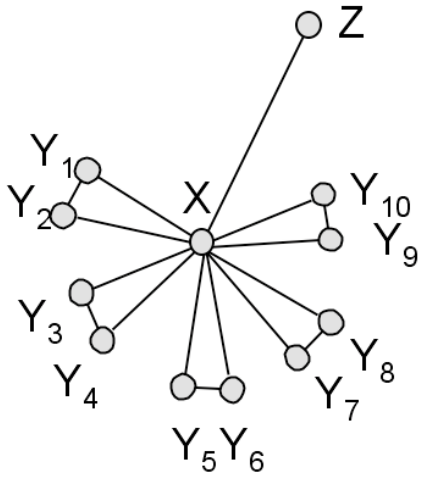

Figure 3.1 A constraint graph for a CSP problem on 12 variables.

Table 3.1 An example of how different preference expression methods impact a single metric. Strengths that express preferences over the available choices in Figure 3.1 are calculated under 4 different methods.

Variables sets (C_i)	X	$Y_1, Y_2, \ldots, Y_{10}$	Z
Degree metric scores (v_i)	**11**	**2**	**1**
Rank strength	3	2	1
Linear strength	3.00	1.20	1.00
Borda-w strength	2.83	1.17	1.00
Borda-wt strength	3.00	2.83	1.17

3.2.1. Ranking

Ranking is ACE's original preference expression method. It determines the strengths of the choices from the orders of the subsets in $\mathscr{C}_A$. Ranking assigns strength p to all choices

in C_1, strength p–1 to all choices in C_2, and so on. For a given Advisor A and the partition $\mathscr{C}_A(C)$ induced by A, the strength that ranking computes for any choice c is

$$rank\_strenght(A,c,C) = \begin{cases} p-k+1 & c \in C_k \text{ and } k \le min(p, |\mathscr{C}_A(C)|) \\ 0 & otherwise \end{cases}$$

Ranking reflects the preference for one choice over another, but it loses information contained in the v_k scores in two ways. First, it ignores the extent to which one choice is preferred over another. For example in Table 3.1, the degrees of variables X and Y_1 differ by 9, while the degrees of Y_1 and Z differ by only 1. Nonetheless, ranking assigns equally spaced strengths (3, 2 and 1) to those variables. Second, ranking ignores how many choices share the same score. For example in Table 3.1, the ranks of choices Y_1 and Z differ by only 1, although the heuristic prefers only one choice over Y_1 and 11 choices over Z. The three approaches in the remainder of this section address those shortcomings.

3.2.2. Linear interpolation

Linear interpolation not only considers the relative position of scores, but also the actual differences between them. Under linear interpolation, strength differences are proportional to score differences. For a given Advisor A, the partition $\mathscr{C}_A(C)$ induced by A, and the number of partitions the Advisor comments on $q = \min(p, |\mathscr{C}_A(C)|)$, the strength that linear interpolation computes for any choice $c \in C_k$ is determined by a linear function on its score v_k through points (v_1, p) and $(v_q, 1)$:

$$linear\_strength(A,c,C) = \begin{cases} p & q = 1 \\ 1 + \dfrac{v_k - v_q}{v_1 - v_q}(p-1) & q > 1 \text{ and } c \in C_k \text{ and } k \le q \\ 0 & otherwise \end{cases} \qquad [3.5]$$

Here, the strengths for choices are real numbers in the interval [1, p]. Choices in C_1 have strength p, choices in C_q have strength 1, and the strengths of the other choices are linearly interpolated from the scores of their underlying metric. For example, in Table 3.1, the strengths are determined by the value of the linear function through the points (11, 3) and (1, 1). Instead of strength 2 for all the *Y* variables, linear interpolation gives them strength 1.2, which is closer to the strength 1 given to variable *Z*, because the degrees of the *Y* variables are closer to the degree of *Z*. The significantly higher degree of variable *X* is reflected in the distance between its strength and those given to the other variable*s*.

3.2.3. Borda methods

The Borda election method was devised by Jean-Charles de Borda in 1770 (Brams and Fishburn, 2002). It inspired the next two preference expression methods. *Borda methods* consider the total number of available choices, the number of choices with a smaller score and the number of choices with an equal score. Thus the strength for a choice is based on its position relative to the other choices. To keep strengths in the same range as those from ranking and linear interpolation, Borda methods normalize by accumulating *points* (the ratio of strength range and the number of choices on which an Advisor comments). For $q = \min(p, |\mathcal{C}_A(C)|)$, the value of point $u(p, \mathcal{C}_A(C))$ is

$$u(p, \mathscr{C}_A(C)) = \frac{p-1}{\sum_{k=1}^{q} |C_k|} \qquad [3.6]$$

For example, in Table 3.1, u = (3–1)/12=0.17. (Note that the denominator is $|C|$ when $p = |\mathscr{C}_A(C)|$, and smaller than $|C|$ when $p < |\mathscr{C}_A(C)|$.)

The first Borda method, *Borda-w*, awards a point for each *win* (metric score higher than the score of some other commented choice). For a given Advisor A, the ordered partition $\mathscr{C}_A(C)$ induced by A, the number of subsets the Advisor comments on $q = \min(p, |\mathscr{C}_A(C)|)$, and the point $u(p, \mathscr{C}_A(C))$ determined by [3.6], the strength *Borda-w* computes for any choice c is

$$Borda-w\_strength(A,c,C) = \begin{cases} 1 & c \in C_q \\ 1 + u(p, \mathscr{C}_A(C)) \cdot \sum_{j=k+1}^{q} |C_j| & c \in C_k \text{ and } k < q \\ 0 & otherwise \end{cases} \qquad [3.7]$$

Borda-w strengths for the example in Figure 3.1 are shown in Table 3.1. The lowest-scoring variable Z has strength 1. Because every Y variable out-scored only Z, the strength of any Y variable is 1+0.17=1.17. The highest-scoring choice X out-scored 11 choices, so X's strength is $1+11 \cdot 0.17=2.83$.

The second Borda method, *Borda-wt,* awards one point for each win and one point for each *tie* (score equal to the score of some other choice). It can be interpreted as

emphasizing each loss. For a given Advisor A, the ordered partition $\mathcal{O}_A(C)$ induced by A, the number of subsets the Advisor comments on $q = \min(p, |\mathcal{O}_A(C)|)$, and the point $u(p, \mathcal{O}_A(C))$ determined by [3.6], the strength *Borda-wt* computes for any choice c is

$$Borda-wt\_strength(A,c,C) = \begin{cases} p & c \in C_1 \\ p - u(p, \mathcal{O}_A(C)) \cdot \sum_{j=1}^{k-1} |C_j| & c \in C_k \text{ and } 1 < k \le q \\ 0 & otherwise \end{cases} \quad [3.8]$$

For example, in Table 3.1, no choice out-scored the highest-scoring choice *A,* so its strength is 3, one choice (*X*) outscored the *Y* variables, so their strengths are reduced by one point (3–0.17=2.83), and 11 choices out-scored *Z*, resulting in strength 3–11*0.17=1.17.

The difference between the two Borda methods is evident when many choices share the same score. *Borda-w* considers only how many choices score lower, so that a large subset results in a big gap in strength between that subset and the previous (more preferred) one. Under *Borda-wt*, a large subset results in a big gap in strength between that subset and the next (less preferred) one. In Table 3.1, for example, the 10 *Y* variables share the same score. Under *Borda-w*, the difference between the strength of any *Y* variable and *X* is 1.66, while the difference between the strength of any *Y* variable and *Z* is only 0.17. Under *Borda-wt*, however, the difference between the strength of any *Y* and *X* is only 0.17, while the difference between the strength of any *Y* and *Z* is 1.66.

3.3. Experimental results

The experiments described here compare the learning and testing performance of DWL and RSWL with different preference expression methods. Full restart was performed with a (5 7) full restart threshold. On *Comp* problems, all 40 Advisors were used at once. In all other experiments, a random selection of 30% of the Advisors was used for each learning problem. The parameter p was 5, hence the Advisors commented on those choices with the $q=min(5, |\mathcal{C}_A(C)|)$ highest scores. Improvements over DWL that were statistically significant at the 95% confidence level appear in bold in the tables.

On the easier *Geo-82* and <30, 8, 0.26, 0.34> classes, no method produced statistically significantly different performance from DWL. (Data omitted.) On <20, 30, 0.444, 0.5> problems, RSWL alone did not provide a statistically significant improvement in the number of nodes during testing, but with RSWL-κ, there was a significant improvement. (Data appear in Table 3.2.) On <50, 10, 0.38, 0.2> problems, there were statistically significant increases in the percentage of solved problems in every

Table 3.2 Learning and testing performance of DWL and RSWL with different preference expression methods on <20, 30, 0.444, 0.5> problems. Statistically significant reductions in the number of nodes, compared to search under DWL, appear in bold.

		Learning			**Testing**	
Weight-learning Algorithm	Preference expression method	Number of problems	Number of unsolved problems	Number of full restarts	Number of nodes	Percentage of solved problems
DWL	Ranking	33.8	6.8	0.4	2,678.6	100.0%
RSWL	Ranking	30.8	5.0	0.1	2,534.5	99.8%
RSWL, h=50%	Ranking	30.8	3.8	0.1	2,427.1	100.0%
RSWL, k=0.5	Ranking	36.6	9.9	0.5	**2,383.9**	100.0%
RSWL	Linear	30.0	5.8	0.0	**2,370.1**	100.0%
RSWL	*Borda-w*	31.5	7.2	0.2	2,471.8	100.0%
RSWL	*Borda-wt*	33.2	8.4	0.3	**2,372.2**	100.0%

case, and often a statistically significant improvement in the number of nodes during testing. (Data appear in Table 3.3.) There was, however, no single best method for both classes.

The random structure of problems from model B classes seems to allow for only limited improvement over the already effective DWL algorithm. Problems with non-random structure may benefit more from preference expression methods, however. On *Comp* problems, in the experiments of Table 3.4, RSWL with every preference expression method improved performance over DWL. Even under sharply reduced node limits during learning, as low as 35 (where a minimum of 30 nodes are needed for a backtrack-free solution), there was no appreciable change in RSWL's performance. Data appear in Table 3.5.

Table 3.3 Learning and testing performance of DWL and RSWL with different preference expression methods on <50, 10, 0.38, 0.2> problems. Statistically significant reductions in the number of nodes and in the percentage of solved problems, compared to search under DWL, appear in bold.

		Learning			**Testing**	
Weight-learning Algorithm	Preference expression method	Number of problems	Number of unsolved problems	Number of full restarts	Number of nodes	Percentage of solved problems
DWL	Ranking	36.8	14.1	0.8	13,708.58	91.8%
RSWL	Ranking	31.5	7.5	0.2	13,111.44	**95.2%**
RSWL, h=30%	Ranking	30.9	8.4	0.1	**11,849.00**	**94.6%**
RSWL, k=0.2	Ranking	32.9	8.9	0.3	**11,231.60**	**95.0%**
RSWL	Linear	30.0	7.1	0.0	13,718.21	**94.2%**
RSWL	*Borda-w*	30.0	7.0	0.0	**12,093.06**	**97.4%**
RSWL	*Borda-wt*	33.2	9.9	0.3	13,098.78	**96.0%**

Table 3.4 Learning and testing performance of DWL and RSWL with different preference expression methods on *Comp* problems. Statistically significant reductions in the number of nodes, compared to search under DWL, appear in bold.

		Learning			**Testing**	
Weight-learning Algorithm	Preference expression method	Number of problems	Number of unsolved problems	Number of full restarts	Number of nodes	Number of problems
DWL	Ranking	30	2.5	0	298.4	95.4%
RSWL	Ranking	30	0.4	0	**161.1**	97.8%
RSWL	Linear	30	0.8	0	**164.2**	97.8%
RSWL	*Borda-w*	30	0.3	0	**139.9**	98.0%
RSWL	*Borda-wt*	30	0.6	0	**151.1**	98.0%

To further emphasize the importance of preference methods on problems with non-random structure, Borda methods were modified to assign some positive strength to all choices (i.e., p is not a constant; instead, $p = |\mathscr{C}_A(C)|$ for every decision). The resultant testing performance appears in Table 3.6. Under this condition, *Borda-w's* performance improved, but the modification made *Borda-wt*'s performance dramatically worse. On *Comp* problems, there are many initial ties in the relatively large central component and its sparsity tends to perpetuate them. As a result, the subsets of choices with the same scores are relatively large as well.

Table 3.5: Testing performance of DWL and RSWL with ranking and Borda preference expression methods with reduced node limits on *Comp* problems.

	DWL		**RSWL-ranking**		**RSWL-*Borda-w***		**RSWL-*Borda-wt***	
Node limit	Number of nodes	Percentage of solved problems	Number of nodes	Percentage of solved problems	Number of nodes	Percentage of solved problems	Number of nodes	Percentage of solved problems
5000	298. 5	95.4%	161.1	97.8%	139.9	98.0%	151.1	98.0%
1000	461.8	92.0%	161.1	97.8%	130.1	98.2%	183.8	97.4%
500	341.9	94.4%	161.5	97.8%	130.5	98.2%	150.7	98.0%
100	495.6	91.2%	161.4	97.8%	139.4	98.0%	190.0	97.4%
35	400.0	93.4%	160.4	97.8%	147.6	98.0%	128.5	98.4%

Table 3.6: Testing performance of RSWL with reduced node limits with modified Borda preference expression methods that assigns strengths to all choices on *Comp* problems.

Node limit	***Borda-w*** comments on all choices		***Borda-wt*** comments on all choices	
	Number of nodes	Percentage of solved problems	Number of nodes	Percentage of solved problems
5000	134.1	98.0%	638.5	88.6%
1000	134.1	98.0%	564.7	89.8%
500	121.1	98.2%	728.5	86.6%
100	111.6	98.4%	642.1	88.2%
35	111.7	98.4%	660.0	89.0%

Under *Borda–wt*, if only a few choices score higher, the strength of the choices from the next lower-scoring subset is close enough to influence the decision. If there are many high-scoring choices, the next lower subset will have much lower strength, which decreases its influence. With *Borda–w*, when many choices share the same score, they are penalized for failure to discriminate, and their strength is lowered. When *Borda–w* assigns lower strengths to large subsets from the central component, it makes them less attractive. That encourages the variables from subproblem defined by the satellite to be selected first; this is often the right way to solve such problems.

Linear extrapolation had performance similar to RSWL on *Comp* problems,

Table 3.7: Testing performance of the linear preference expression method with reduced node limits on *Comp* problems.

RSWL-linear		
Node limit	Number of nodes	Percentage of solved problems
5000	164.17	97.8%
1000	164.17	97.8%
500	164.17	97.8%
100	163.71	97.8%
35	33.52	100.0%

except that under the lowest node limit, 35 nodes, RSWL with linear preference expression was able to solve every problem during testing. The 35-node limit imposes a very high learning standard; it allows learning only from perfect or almost perfect solutions. Only with the nuances of information provided by linear preference expression did ACE develop a weight profile that solved all the testing problems in every run.

4. Chapter 4

4.1. Results

Full restart is a novel strategy that accelerates the convergence of learning to a high-performance combination of heuristics, even though the optimal combination and the distance to it are unknown. Previously, restart was used either to make a fresh start on global search for solution to a single problem, or to make local search more resilient to local minima (Russell and Norvig, 2003). In both cases, restart was meant to find a solution to a problem more quickly, not to improve learning performance. When used to solve difficult CSPs, Rapid Randomized Restart effectively eliminates the right heavy tail (frequent extremely long solutions) and benefits from the left heavy tail (frequent extremely short solutions) observed in the run time distribution of a fixed algorithm search on individual problems (Gomes, Selman, Crato, et al., 2000). Randomized restart of search has successfully been applied to Internet traffic, scheduling, theorem proving, circuit synthesis, planning, and hardware verification (Kautz, Horvitz, Ruan, et al., 2002; Sadeh-Koniecpol, Nakakuki and Thangiah, 1997).

Full restart helps the learner respond, early on, to learning that is not going well. It is beneficial across the range of node limits. Under lower node limits and without full restart, if initially class-inappropriate Advisors garner high weights, the weights of class-appropriate Advisors cannot recover. Under higher node limits, weights can eventually recover without the use of full restart, but recovery is more expensive.

Given an initial set of heuristics that is large and inconsistent, many class-inappropriate heuristics may combine to make bad choices, and thereby make it difficult

to solve the problem within a given node limit. Because only solved problems provide training instances for weight learning here, no learning can take place until some problem is solved. Rather than consult all its Advisors at once, ACE can randomly select a new subset of Advisors for each problem, consult them, make decisions based on their comments, and update only their weights. This method, learning with random subsets, eventually uses a subset in which class-appropriate heuristics predominate and agree on choices that solve a problem.

Learning with random subsets improves performance: there are fewer failures to solve a problem within the given node limit, decisions are faster during learning, and more problems are solved with fewer nodes during testing. Learning with random subsets is shown here to be a robust method that works well with different subset sizes and with different proportions of class-appropriate to class-inappropriate heuristics. Without random subsets, under its available resources, and with approximately an equal number of class-appropriate and class-inappropriate Advisors, ACE struggled to learn on hard problem classes. When class-inappropriate heuristics predominated in the pool of heuristics, only with random subsets was any learning possible at all.

Learning time may be less important, since learning happens only once, but in practice, it is certainly relevant. Learning with a large number of heuristics is computationally expensive. Reducing an initial pool of heuristics without the identification and elimination of those irrelevant to a given problem class risks overlooking exceptionally good heuristics for a given class of problems. With random subsets, the reduced number of heuristics consulted for each problem significantly reduces the computational effort during learning, while the pool of heuristics remains

large.

RSWL is a weight-learning algorithm that reinforces weights based upon the distribution of each heuristic's preferences across all the available choices and upon an estimate of how difficult it is to make the correct decision. For example, an Advisor that strongly singles out the correct decision in a positive training instance receives more credit than a less-discriminating Advisor, and an Advisor that supports the wrong choice among a few possible choices receives a harsh penalty. A preference expression method can exploit heuristics' comparative opinions in ways that consider both the scores returned by the metrics on which these heuristics rely and the distributions of those scores across a set of possible choices. The simplest way to combine heuristics' preferences is mere ranking of these scores, to scale them into some common range. Linear interpolation not only considers the relative position of choices, but also the actual differences between scores by making strength differences proportional to the differences in scores. The Borda methods emphasize the relative position of a choice among other choices, and attend to how many choices share the same score. The variations used here set strength based on the number of choices scored lower than this choice or not higher than this choice. No single one method to combine preferences outperformed ranking on every problem class, but each was better on some individual classes.

4.2. Significance

A good weighted combination of ordering heuristics can significantly improve search to solve CSP problems. Other solvers can use a weight profile produced by ACE for given problem class. This particular work addresses problems with non-random structure

because they are particularly challenging for traditional CSP solvers. Indeed, both *Comp* and some geometric problems appeared in the First International Constraint Solver Competition at Principles and Practice of Constraint Programming (CP-2005).

The techniques presented in this work are demonstrated on ordering heuristics for solving CSPs, but they are general and applicable to other machine learning domains. Those algorithms can be used for problems in any environment where heuristics are used without direct supervision to search for a solution. Learning with random subsets is a general technique that can be used to extract some class-specific knowledge. Preference expression methods can be modified to find a mixture when the comparative opinions of different experts are available.

Because a good mixture of heuristics is effectively a new heuristic, this research contributes to extensive AI research on developing new heuristics for solving NP-complete problems. Another important AI research area is adaptive search. ACE is a solver that gauges and adapts its search procedure (Petrovic, Epstein and Wallace, 2007). Full restart monitors learning progress and takes an action if it is not satisfactory. RSWL with different preference evaluation methods adapts ACE's decision making, its reinforcement policy, and its heuristic selection by evaluating new parameters such as the accuracy, intensity, frequency and distribution of its heuristics' preferences.

Planning is another AI area that can benefit from this research. When a planning problem is cast as a constraint satisfaction problem, it can use the representational expressiveness and propagation power inherent in constraint programming (Nareyek, Freuder, Fourer, et al., 2005). If such an encoding lacks the necessary planning knowledge, the methods presented here could help to learn effective solution methods

(Epstein and Petrovic, 2007).

4.3. Future work

I plan to extend this research to other kinds of problems. Results are demonstrated here on classes of solvable, binary CSPs with random and nonrandom structure. I plan to use classes of problems with non-binary constraints, and classes that contain unsolvable problems. Particularly interesting are optimization problems and real-world problems.

An important part of my future research is how to determine good values for learning parameters, to better understand interaction among them, and to evaluate strategies that would learn and dynamically update their values. For example, the reducing node limit during learning would impose higher standards once some learning occurs. At that point, the termination criteria for learning, full restart threshold and random subsets sizes should accordingly change.

The appropriate restart cutoff value for an individual problem has been extensively studied. If the runtime distribution of a problem is known, it is possible to compute an optimal fixed cutoff value; if the distribution is unknown, there is a universal strategy provably within a log factor of optimal (Luby, Sinclair and Zuckerman, 1993). Another successful strategy increases the cutoff value geometrically (Walsh, 1999). When even partial knowledge of the effort distribution is known, and data on the search process is available, an appropriate restart cutoff can be dynamically determined (Kautz, Horvitz, Ruan, et al., 2002). Still another approach dynamically identifies problem features that underlie runtime distribution. It uses them to partition problem instances into classes with smaller runtime variability, with a different cutoff for each subset (Ruan, Horvitz and Kautz, 2003). That research can be applied to the full restart threshold.

My current work seeks to determine a good random subset size. The robustness of learning with different sizes of random subsets is demonstrated here, but the benefits of random subsets can further be increased by appropriate subset sizes. As learning progresses, more information can provide insight about the pool of heuristics and the degree of randomness can be accordingly reduced. Random subsets are an important tool to launch learning, and less beneficial when class-appropriate heuristics are identified and learning is used for the finer adjustments on weights. When the ratio of class-appropriate to class-inappropriate heuristics is small, it is important to use smaller subsets; monitoring the frequency of unsolved problems may provide insight on how to adapt random subsets sizes.

Another important parameter in learning is the correlation among heuristics (Wallace, 2005; Wallace, 2006; Wallace and Bain, 2007). If the opinion of an Advisor is represented as a vector containing strengths of recommendation for each of available choices, then the difference between two Advisors can be defined as a distance between the two vectors (e.g., Euclidean distance or Manhattan distance.) If a group of heuristics' vectors are often close on given class of problems, some of those heuristics can be eliminated from the pool of heuristics (e.g., a group can be replaced by its representative) or they may receive special treatment when random subsets are used. That future research will include how to determine the elimination criteria (e.g., current weight, computational cost, some measure of impact on other heuristics) and how to incorporate the weights of the eliminated heuristics in the weight of their remaining representative.

Correlation among heuristics depends not only on the class of problems, but also on the method used to express the heuristics' preferences. This thesis demonstrates that

preference can be used to improve learning, particularly on problems with non-random structure, but not how to choose a preference method. Correlation among heuristics under different preference methods on a given class of problems may provide insight on how to select a preference method.

Appendix A: List of Advisors used in this work

The following Advisors were used in these experiments.

Tier-1 Advisors:

- ***Victory.*** When only a single variable has no assigned value and has been selected, Victory comments in favor of any value in the dynamic domain of that variable.
- ***Degree zero.*** When a variable is to be selected next, Degree Zero vetoes any variable whose dynamic degree is zero.
- ***Unique value.*** When a variable is to be selected next, Unique Value forces the selection of any variable whose dynamic domain contains exactly one value.

Tier-3 Advisors:

The concerns underlying ACE's tier-3 Advisors are drawn from the CSP literature. (Citations are provided where possible.) Two vertices with an edge between them are *neighbors*. A *nearly neighbor* is the neighbor of a neighbor in the constraint graph. The *degree of an edge* is the sum of the degrees of the variables incident on it. The *edge degree of a variable* is the sum of edge degrees of the edges on which it is incident. Advisors are listed in dual pairs.

Variable-ordering Advisors:

- ***Min/Max degree*** supports variables in increasing/decreasing order of their static (in the original constraint graph) degree.
- ***Min/Max domain*** supports variables in increasing/decreasing order of their dynamic domain size after propagation.

- ***Min/Max domain/static-degree*** supports variables in increasing/decreasing order of the ratio of their dynamic domain size to their static degree (Bessière and Régin, 1996).
- ***Min/Max backward-degree*** supports variables in increasing/decreasing order of the number of their valued neighbors.
- ***Min/Max dynamic-degree*** supports variables in increasing/decreasing order of their dynamic degree (number of unvalued neighbors).
- ***Min/Max value-pairs*** supports variables in increasing/decreasing order of the number of pairs of values with their neighbors still supported by the current partial assignment (Kiziltan, Flener and Hnich, 2001).
- ***Min/Max static-connected-edges*** orders the edges in the original constraint graph descendingly, by the sum of the degrees of their vertices. They support variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.
- ***Min/Max static-less-connected-edges*** orders the edges in the original constraint graph ascendingly, by the sum of the degrees of their vertices. They support variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.
- ***Min/Max dynamic-connected-edges*** orders the edges in the dynamic constraint graph descendingly, by the sum of the degrees of their vertices. Supports variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.

- ***Min/Max dynamic-less-connected-edges*** orders the edges in the dynamic constraint graph ascendingly, by the sum of the degrees of their vertices. Supports variables in increasing/decreasing order of their incidence on these edges, with preference for the higher-degree vertex.
- ***Min/Max FF2*** supports variables in increasing/decreasing order of the estimate of their likelihood of failure, computed for variable v_i by

$$\left(1 - \prod_{i \neq j} \left(1 - {t_{ij}}^{m_j}\right)\right)^{m_j}$$

  where t_{ij} denotes the current tightness of the constraint between i and j (or zero if there is none), and m_j is the static degree of the variable (Smith and Grant, 1998)
- ***Min/Max weighted-degree*** supports variables in increasing/decreasing order of the sum of the weights of the edges on which they are incident. Initially, edge weights are set to 1. Thereafter, each time propagation from one endpoint of an edge wipes out the domain of the other endpoint, the weight of the edge is increased by 1 (Boussemart, Hemery, Lecoutre, et al., 2004).
- ***Min/Max domain/weighted-degree*** supports variables in increasing/decreasing order of the ratio of their dynamic domain size to their weighted degree (Boussemart, Hemery, Lecoutre, et al., 2004).

Value-ordering Advisors:

- ***Min/Max static-conflicts*** *Value* minimizes/maximizes based on the number of values that would be supported in the static domains of the unvalued neighbors of the variable (Frost and Dechter, 1995).

- ***Min/Max small-domain-value*** minimizes/maximizes the minimal domain size among the neighbors of the variable after this assignment (Frost and Dechter, 1995).
- ***Min/Max product-domain-value*** minimizes/maximizes the product of the domain sizes of the neighbors of the variable after this assignment
- ***Min/Max domain-score-value*** minimizes/maximizes the largest domain size of the neighbors of the variable after this assignment. It takes the number of unvalued variables with domains of that size as an exponent.
- ***Min/Max secondary-pairs-value*** minimizes/maximizes the number of value pairs supported by this assignment from neighbors of the variable to nearly neighbors of the variable.
- ***Min/Max secondary-value*** minimizes/maximizes the number of values supported by this assignment among nearly neighbors of the variable.

Appendix B: Sample weights of the Advisors

Table B.1: Weights of variable ordering Advisors from successful runs that produced different performance and from an inadequate run on problems in <30, 8, 0.26, 0.34> class.

Advisors	Weights in a run with average number of nodes 100.76	Weights in a run with average number of nodes 145.88	Sample weights in an inadequate run
Benchmark-variable	1.53	1.00	0.77
Max-static-degree	2.21	1.36	0.13
Min-static-degree	0.89	0.38	2.24
Max-domain	0.05	0.00	0.00
Min-domain	1.55	0.00	0.00
Min-FF2.2	0.18	0.22	2.49
Max-FF2.2	4.01	1.07	0.10
Max-backward-degree	1.14	0.95	0.31
Min-backward-degree	1.75	0.46	0.84
Max-dynamic-degree	3.58	1.07	0.12
Min-dynamic-degree	0.15	0.24	2.29
Min-value-pairs	0.73	0.91	2.41
Max-value-pairs	2.32	1.35	0.12
Min-static-connected-edges	0.36	0.91	2.80
Max-static-connected-edges	2.62	1.13	0.41
Max-static-less-connected-edges	1.67	1.59	1.88
Min-static-less-connected-edges	1.11	0.91	0.36
Max-dynamic-connected-edges	3.24	1.73	0.41
Min-dynamic-connected-edges	0.39	0.45	2.49
Max-dynamic-less-connected-edges	1.43	0.68	1.61
Min-dynamic-less-connected-edges	2.01	1.36	0.44
Min-domain/static-degree	2.61	1.36	0.10
Max-domain/static-degree	0.85	0.36	2.37
Min-domain/dynamic-degree	4.17	1.06	0.10
Max-domain/dynamic-degree	0.17	0.23	2.07
Min-weighted-degree	0.42	0.00	2.45
Max-weighted-degree	3.17	1.19	0.28
Min-dynamic-domain/weighted-degree	3.74	1.19	0.28
Max-dynamic-domain/weighted-degree	0.41	0.00	2.62

Table B.2: Weights of value ordering Advisors from successful runs that produced different performance and from an inadequate run on problems in <30, 8, 0.26, 0.34> class.

Advisors	Weights in a run with average number of nodes 100.76	Weights in a run with average number of nodes 145.88	Sample weights in an inadequate run
Benchmark-value	0.09	0.12	0.00
Min-static-conflicts-value	0.26	0.00	1.09
Max-static-conflicts-value	0.09	0.26	0.00
Max-small-domain-value	0.37	0.00	0.00
Min-small-domain-value	0.00	0.00	0.00
Max-product-domain-value	0.42	0.12	0.60
Min-product-domain-value	0.00	0.00	0.00
Max-secondary-pairs-value	0.05	0.00	0.60
Min-secondary-pairs-value	0.09	0.11	0.35
Max-secondary-value	0.23	0.00	0.17
Min-secondary-value	0.02	0.15	0.00
Max-domain-score-value	0.07	0.19	0.20
Min-domain-score-value	0.34	0.00	0.54

Appendix C: Variable-ordering Advisors used in Section 2.4

Table C.1: Variable-ordering Advisors used for experiments in the Table 2.10. These sets are deliberately biased to make learning more difficult.

	Class-appropriate Advisors	**Class-inappropriate Advisors**
Set-1	*Max-static-degree* *Min-domain* *Min-domain/dynamic-degree* *Max-weighted-degree* *Min-dynamic-domain/weighted-degree* *Min-static-less-connected-edges*	*Min-static-degree* *Max-domain* *Max-domain/dynamic-degree* *Min-weighted-degree* *Max-dynamic-domain/weighted-degree* *Min-static-connected-edges* *Min-backward-degree* *Min-dynamic-degree* *Max-domain/static-degree*
Set-2	*Max-static-degree* *Min-domain* *Max-backward-degree* *Max-dynamic-degree* *Min-dynamic-domain/weighted-degree* *Min-static-less-connected-edges*	*Min-static-degree* *Max-domain* *Min-backward-degree* *Min-dynamic-degree* *Max-dynamic-domain/weighted-degree* *Min-static-connected-edges* *Max-domain/static-degree* *Max-domain/dynamic-degree* *Min-weighted-degree*
Set-3	*Max-static-degree* *Min-domain* *Min-dynamic-domain/weighted-degree* *Min-static-less-connected-edges*	*Min-static-degree* *Max-domain* *Max-dynamic-domain/weighted-degree* *Min-static-connected-edges* *Min-backward-degree* *Min-dynamic-degree* *Max-domain/static-degree* *Max-domain/dynamic-degree* *Min-weighted-degree*

Bibliography

1. Aardal, K. I., S. P. M. v. Hoesel, A. M. C. A. Koster, C. Mannino and A. Sassano (2003). Models and solution techniques for frequency assignment problems. *4OR: A Quarterly Journal of Operations Research* 1(4): 261-317.

2. Aha, D. W., D. F. Kibler and M. K. Albert (1991). Instance-Based Learning Algorithms. *Machine Learning* 6: 37-66.

3. Ali, K. and M. Pazzani (1996). Error reduction through learning multiple descriptions. *Machine Learning* 24: 173-202.

4. Bellman, R. E. (1957). Dynamic Programming, Princeton University Press (Republished 2003).

5. Bessière, C. and J.-C. Régin (1996). MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. *Principles and Practice of Constraint Programming CP-96* (E. E. E. Freuder, Ed.), pp. 61-75, Springer-Verlag

6. Borrett, J. E., E. P. K. Tsang and N. R. Walsh (1996). Adaptive Constraint Satisfaction: The Quickest First Principle. *European Conference on Artificial Intelligence*, pp. 160-164.

7. Boussemart, F., F. Hemery, C. Lecoutre and L. Sais (2004). Boosting systematic search by weighting constraints. *Sixteenth European Conference on Artificial Intelligence-ECAI'04*, pp. 146–150.

8. Brams, S. J. and P. C. Fishburn (2002). Voting procedures. *Handbook of Social Choice and Welfare* Volume 1: 173-236.

9. Buntine, W. (1991). Learning classification trees. *Artificial Intelligence Frontiers in Statistics*, pp. 182--201.

10. Cheeseman, P., B. Kanefsky and W. M. Taylor (1991). Where the really hard problems are. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, {IJCAI}-91*, pp. 331-337, Sidney, Australia.

11. Dechter, R. (2003). Constraint Processing, Morgan Kaufmann, San Francisco, CA.

12. Dietterich, T. G. (2000). Ensemble methods in machine learning. *First International Workshop on Multiple Classifier Systems*, pp. 1-15, Cagliari, Italy.

13. Dietterich, T. G. (2003). Machine Learning. In Nature Encyclopedia of Cognitive Science, Macmillan, London.

14. Do, M. B. and S. Kambhampati (2001). Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence* 132: 151-182.

15. Epstein, S. L. (1994). For the Right Reasons: The FORR Architecture for Learning in a Skill Domain. *Cognitive Science* 18: 479-511.

16. Epstein, S. L. (1995). On Heuristic Reasoning, Reactivity, and Search. *Fourteenth International Joint Conference on Artificial Intelligence*, pp. 454-461, Morgan Kaufmann, Montreal.

17. Epstein, S. L. (2004a). A Cognitively Oriented Architecture Confronts Hard Problems. *Proceedings of the AAAI Fall Symposium on Achieving Human-Level Intelligence* AAAI-2004.

18. Epstein, S. L. (2004b). Metaknowledge for Autonomous Systems. *Proceedings of AAAI Spring Symposium on Knowledge Representation and Ontology for Autonomous Systems. AAAI*, pp. 61-68.

19. Epstein, S. L. and E. Freuder (2001). Collaborative Learning for Constraint Solving. *Principles and Practice of Constraint Programming -- CP2001* (T. Walsh, Ed.), pp. 46 - 60, Springer 2001, Paphos, Cyprus.

20. Epstein, S. L., E. C. Freuder and R. Wallace (2005). Learning to Support Constraint Programmers. *Computational Intelligence* 21(4): 337-371.

21. Epstein, S. L., E. C. Freuder, R. Wallace, A. Morozov and B. Samuels (2002). The Adaptive Constraint Engine. *Principles and Practice of Constraint Programming -- CP 2002, 8th International Conference* (P. V. Hentenryck, Ed.), pp. 525 - 542, Springer 2002, Ithaca, NY, USA.

22. Epstein, S. L. and S. Petrovic (2007). Learning to Solve Constraint Problems. *Workshop on AI Planning and Learning, International Conference on Automated Planning and Scheduling, ICAPS-07*, Providence, Rhode Island, USA.

23. Freuder, E. C. (1978). Synthesizing Constraint Expressions. *Communications of the ACM* 21: 958 - 965.

24. Freuder, E. C. (1982). A sufficient condition for backtrack-free search. *Journal of the ACM* 29: 24-32.

25. Freund, Y. and R. Schapire (1996). Experiments with a new boosting algorithm. *Thirteenth International Conference on Machine Learning*, pp. 148-156.

26. Friedman, N., D. Geiger and M. Goldszmidt (1997). Bayesian network classifiers. *Machine Learning* 29: 131--163.

27. Frost, D. and R. Dechter (1995). Look-ahead Value Ordering for Constraint Satisfaction Problems. *IJCAI-95*, pp. 572-278.

28. Fukunaga, A. S. (2002). Automated Discovery of Composite SAT Variable-Selection Heuristics. *AAAI/IAAI*, pp. 641-648.

29. Gagliolo, M. and J. Schmidhuber (2006). Dynamic Algorithm Portfolios. In Ninth International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida.

30. Gent, I. P., P. Prosser and T. Walsh (1996). The Constrainedness of Search. *AAAI/IAAI* 1: 246-252.

31. Gomes, C., C. Fernandez, B. Selman and C. Bessière (2004). Statistical Regimes Across Constrainedness Regions. *10th Conf. on Principles and Practice of Constraint Programming (CP-04)* (M. Wallace, Ed.), pp. 32-46, Springer, Toronto, Canada.

32. Gomes, C. P. and B. Selman (2001). Algorithm portfolios. *Artificial Intelligence* 126: 43-62.

33. Gomes, C. P., B. Selman, N. Crato and H. Kautz (2000). Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*: 67–100.

34. Hansen, L. and P. Salamon (1990). Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12: 993-1001.

35. Haralick, R. M. and G. L. Elliott (1980). Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence* 14: 263-313.

36. Harmon, M. E. and S. S. Harmon (1997). Reinforcement Learning: A Tutorial. http://citeseer.ist.psu.edu/harmon96reinforcement.html.

37. Hulubei, T. and B. O'Sullivan (2005). Search Heuristics and Heavy-Tailed Behavior. *Principles and Practice of Constraint Programming - CP 2005* (P. V. Beek, Ed.), pp. 328-342, Berlin: Springer-Verlag.

38. Johnson, D. S., C. R. Aragon, L. A. McGeoch and C. Schevon (1989). Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research* 37: 865-892.

39. Kautz, H., E. Horvitz, Y. Ruan, C. Gomes and B. Selman (2002). Dynamic restart policies. *Eighteenth National Conference on Artificial Intelligence*, pp. 674 - 681, AAAI Press, Edmonton, Alberta, Canada.

40. Kivinen, J. and M. K. Warmuth (1999). Averaging expert predictions. *Computational Learning Theory: 4th European Conference (EuroCOLT '99)*, pp. 153--167, Springer, Berlin.

41. Kiziltan, Z., P. Flener and B. Hnich (2001). Towards Inferring Labelling Heuristics for CSP Application Domains. *Joint German/Austrian Conference on AI: Advances in Artificial Intelligence*, pp. 275 - 289, Lecture Notes In Computer Science; Vol. 2174.

42. Lamb, D. and A. Bandopadhay (1993). Shape from line drawings: beyond Huffman-Clowes labeling. *Pattern Recognition Letters* 14: 213-219.

43. Lecoutre, C., F. Boussemart and F. Hemery (2004). Backjump-based techniques versus conflict directed heuristics. *ICTAI*: 549–557.

44. Luby, M., A. Sinclair and D. Zuckerman (1993). Optimal Speedup of Las Vegas Algorithms. *Israel Symposium on Theoretical Aspects of Computer Science ISTCS*, pp. 128-133.

45. Minton, S., J. A. Allen, S. Wolfe and A. Philpot (1995). An Overview of Learning in the Multi-TAC System. *First International Joint Workshop on Artificial Intelligence and Operations Research*, Timberline, Oregon, USA.

46. Mitchell, T. (1997). Machine Learning, McGraw Hill, New York.

47. Nareyek, A. (2004). Choosing Search Heuristics by Non-Stationary Reinforcement Learning. *Metaheuristics: Computer Decision-Making*: 523-544.

48. Nareyek, A., E. C. Freuder, R. Fourer, E. Giunchiglia, R. P. Goldman, H. Kautz, J. Rintanen and A. Tate (2005). Constraints and AI planning. *IEEE Intelligent Systems* 20: 62 - 72.

49. Opitz, D. and J. Shavlik (1996). Generating accurate and diverse members of a neural-network ensemble. *Advances in Neural Information Processing Systems* 8: 535-541.

50. Otten, L., M. Grönkvist and D. P. Dubhashi (2006). Randomization in Constraint Programming for Airline Planning. *Principles and Practice of Constraint Programming CP-2006*, pp. 406-420, Nantes, France.

51. Papadimitriou, C. H. (1994). Computational Complexity, Addison-Wesley Publishing.

52. Petrie, K. E. and B. M. Smith (2003). Symmetry breaking in graceful graphs. In Principles and Practice of Constraint Programming CP-2005, pp. 930-934, LNCS 2833.

53. Petrovic, S. and S. L. Epstein (2006a). Full Restart Speeds Learning. *Proceedings of the 19th International FLAIRS Conference (FLAIRS-06)*, Melbourne Beach, Florida.

54. Petrovic, S. and S. L. Epstein (2006b). Relative Support Weight Learning for Constraint Solving. *AAAI Workshop on Learning for Search*, pp. 115-122, Boston.

55. Petrovic, S. and S. L. Epstein (2007a). Preferences Improve Learning to Solve Constraint Problems. *AAAI Workshop on Preference Handling for Artificial Intelligence*, pp. 71-78, Vancouver, Canada.

56. Petrovic, S. and S. L. Epstein (2007b). Random Subsets Support Learning a Mixture of Heuristics. *Proceedings of the 20th International FLAIRS Conference (FLAIRS-07)*, Key West, Florida.

57. Petrovic, S. and S. L. Epstein (In press). Random Subsets Support Learning a Mixture of Heuristics. *International Journal on Artificial Intelligence Tools (IJAIT)*.

58. Petrovic, S., S. L. Epstein and R. J. Wallace (2007). Learning a Mixture of Search Heuristics. *In Proceedings of Workshop on Autonomous Search, Principles and Practice of Constraint Programming, CP-07*, Providence, Rhode Island, USA.

59. Prestwich, S. (2008). Generalised graph colouring by a hybrid of local search and constraint programming. *Discrete Applied Mathematics* 156: 148-158.

60. Ruan, Y., E. Horvitz and H. Kautz (2003). Hardness-Aware Restart Policies. *IJCAI-03 Workshop on Stochastic Search Algorithms*, Acapulco, Mexico.

61. Ruan, Y., H. Kautz and E. Horvitz (2004). The backdoor key: A path to understanding problem hardness. *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pp. 124-130, San Jose, CA, USA.

62. Russell, S. and P. Norvig (2003). Artificial Intelligence A Modern Approach, Prentice Hall, Upper Saddle River, NJ.

63. Sabin, D. and E. C. Freuder (1997). Understanding and Improving the MAC Algorithm. *Principles and Practice of Constraint Programming*: 167-181.

64. Sadeh, N. and M. S. Fox (1996). Variable and Value Ordering Heuristics for the Job Shop Scheduling. *Artificial Intelligence* 86: 1-41.

65. Sadeh-Koniecpol, N., Y. Nakakuki and S. R. Thangiah (1997). Learning to Recognize (Un)Promising Simulated Annealing Runs: Efficient Search Procedures for Job Shop Scheduling and Vehicle Routing. *Annals of Operations Research* 75: 189-208.

66. Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning* 5(2): 197--227.

67. Schmitt, L. M. (2001). Theory of Genetic Algorithms. *Theoretical Computer Science* 259: 1-61.

68. Smith, B. and S. Grant (1998). Trying Harder to Fail First. *European Conference on Artificial Intelligence*, pp. 249-253.

69. Streeter, M., D. Golovin and S. F. Smith (2007). Combining multiple heuristics online. *Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, pp. 1197–1203.

70. Sutton, R. S. and A. G. Barto (1998). Reinforcement Learning: An Introduction, MIT Press, Cambridge, Massachusetts.

71. Valentini, G. and F. Masulli (2002). Ensembles of learning machines. *Neural Nets WIRN Vietri-02* (M. M. a. R. Tagliaferri, Ed.), Springer-Verlag, Heidelberg, Italy.

72. Wallace, R. J. (2005). Factor analytic studies of CSP heuristics. *Principles and Practice of Constraint Programming - CP 2005* (P. v. Beek, Ed.), pp. 712-726, Springer.

73. Wallace, R. J. (2006). Analysis of heuristic synergies. In Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming.

74. Wallace, R. J. and S. Bain (2007). Branching Rules for Satisfiability Analysed with Factor Analysis. *Australian Conference on Artificial Intelligence*, pp. 803-809.

75. Walsh, T. (1999). Search in a small world. *16th International Joint Conference on Artificial Intelligence IJCAI'99*, pp. 1172--1177, Morgan Kaufmann Publishers, San Francisco, CA, Stockholm, Sweden.

76. White, H. (1989). Learning in Artificial Neural Networks: A Statistical Perspective. *Neural Computation* 1: 425-464.

77. Young, H. P. (1988). Condorcet's theory of voting. *The American Political Science Review* 82: 1231-1244.