CHAPTER 4

Theory Based Estimates for *ZDC* Formulation Evaluation Heuristics

In chapter 3 we described a context for the heuristic selection of *ZDC* formulations. As part of that work we identified one major area of research which is yet to be explored, namely the development of effective heuristics for evaluating the relative merits of different *ZDC* formulations of a problem. These heuristics are known as evaluation heuristics, H_e, and their main function is to resolve vectors of properties which reflect the expected search cost of individual *ZDC* formulations.

One property which we believe has a significant role to play in the composition of any vector of measures is that of estimated complexity of a *ZDC* formulation. This property has been considered by several researchers in the past, with differing motivations. For example, Knuth (Knuth 1975) uses a monte-carlo based technique for estimating the size of a backtrack tree. In a similar way, Purdom (Purdom 1978) describes an algorithm for predicting search costs. Their motivation was to obtain a view on how long a particular search was likely to take. The approaches of both Knuth and Purdom involve a degree of sampling of the search space.

Nadel describes an alternative approach by developing a selection of theoretically derived equations for estimating the search costs of CSPs (Nudel 1982, 1983a, 1983b) (Nadel 1990a, 1990b, 1995). His motivation covered various aspects of CSP solving including the prediction of

search costs, algorithm selection and the design of effective variable ordering heuristics. Importantly, Nadel was also the first researcher to suggest that this approach could be used for comparing the relative merits of alternative *ZDC* formulations of a problem (Nadel 1990a).

The work by Nadel on comparing alternative representations of a problem was significant, but preliminary in nature. His work looked solely at different *ZDC* formulations of the *n*-Queens problem, using values 3, 4 and 5 for *n*. His approach was also only applied to two algorithms. However, it did provide a useful measure, based on theory, which is relatively cheap to compute and which, unlike the approaches of Knuth and Purdom, involves no sampling of the search space.

In this chapter we make two important contributions. First we demonstrate how Nadel's work can be applied to other algorithms by extending his theories to the backjumping algorithm (Gaschnig 1979). This is important because backjumping belongs to the class of intelligent backtracking algorithms which have been shown to provide improvements in search efficiency (Kondrak & van Beek 1995). Our second contribution is to carry out an extensive evaluation of Nadel's approach, as well as our extension of it. This has not been done previously and so our work provides other researchers with a greater understanding of the applicability of the approach. Together, these contributions demonstrate how theoretical complexity estimates have a major role to play in the design of *ZDC* formulation evaluation heuristics.

In the next section we give an overview of previous work relating to theoretical complexity estimates. In section 4.2 we detail the important aspects of Nadel's approach and then in section 4.3 we present our extension to it. Sections 4.4 and 4.5 give the details of our evaluation of the theoretical complexity approach. The chapter is then concluded with a discussion of our work.

4.1 Probability Models for Constraint Satisfaction Problems

One of the first attempts to develop a probabilistic model for constraint satisfaction problems was presented in (Haralick & Elliott 1980). Haralick and Elliott devised a very simple model for problem classes defined by *K(n, m)* where *n* is the number of variables and *m* is the uniform domain size of all *n* variables. They also assume that for a given pair of variables any pair of labellings to them is consistent with probability *p*, *p* being independent of which variables or which labels. Using this model, Haralick and Elliott developed equations for estimating the number of

solutions to CSPs defined by a given class. They also derived expressions for the expected number of compatibility checks and the expected number of nodes expanded for the standard backtracking and forward checking algorithms. A summary of some of their most important expressions is given in equations (4-1) - (4-3).

The expected number of solutions for class *K(n, m)* is effectively given by the product of search space complexity and probability that all the constraints between all the variables are satisfied. This gives;

$$N(Solutions) = m^n p^{n(n-1)/2} \tag{4-1}$$

where *p* is the probability of compatibility between any pair of labels.

For the standard backtracking algorithm, the expected number of nodes expanded at level *k* is found by multiplying the number of possible labellings at that level and the probability of satisfying all constraints in previous search levels. This gives us;

$$N_k = m^k p^{(k-1)(k-2)/2} \tag{4-2}$$

The expected number of consistency checks at level *k* for standard backtracking is then found by multiplying the expected number of nodes at level *k* by the expected number of constraint checks that are likely to be performed for each node ;

$$C_k = m^k p^{(k-1)(k-2)/2} \left(\frac{1 - p^{k-1}}{1 - p} \right) \tag{4-3}$$

Haralick and Elliott also present corresponding equations for the forward checking algorithm in (Haralick & Elliott 1980).

It is clear that the class of problems defined by *K(n, m)* is very restricted in its scope. For example, many CSPs have different domain sizes for different variables and Haralick & Elliott's model cannot be applied to these. We also often find different compatibilities between different labellings of different variables. However, Haralick and Elliott's results are significant since their analysis does allow a theory-based comparison of the nature of the performance of different algorithms. This was demonstrated with theory-based analysis of the contrasting search profiles of forward checking and standard backtracking algorithms (Haralick & Elliott 1980).

Nadel's early research (Nudel 1982, 1983a, 1983b) was concerned with improving the ideas presented by Haralick and Elliott in terms of the resolution of problem class being considered. He observes that;

Observation 4.1: The more detail one chooses to ignore about CSPs, the coarser is the corresponding partition and the easier it is to carry out a (worst case or expected) complexity analysis over a generic equivalence class - *but* the less relevant it becomes for an individual problem.

One of Nadel's prime motivations was to be able to make estimates of the expected complexity of solving individual CSPs.

Nadel develops a hierarchy of probability models which increase in their level of refinement. He refers to the model of Haralick and Elliott as the *level-0* model (Nudel 1983a). His first refinement was to allow variables to have different domain sizes and to allow for different probabilities of satisfaction to exist between different pairs of variables. This gave us the *level-1* model which applies to problem classes *K(n, **m**)*, where ***m*** is a vector of domain sizes for the individual variables. The level-1 model also relaxes the restriction on the probability of compatibility of any given pair of labels being fixed at *p*. Instead, we have a compatibility between the values of any pair of variables, *i* and *j*, defined to be p_{ij}.

A further refinement leads to the *level-2* model. This can be applied to problem classes *K(n, **m**, [I_{ij}])*. Here the additional parameter defining the class, I_{ij}, is a matrix of the counts of compatible

labels between any given pair of variables, *i* and *j*. The addition of this parameter allows us to define classes of CSPs to the level of individual constraints between variables.

A summary of the different models is given in Table 1. It is based on the overview given in (Nudel 1983a).

Model Level	CSP Domain	Satisfiability Model[1]
0	$K(n, m)$	Prob(T_{ijkl}=1) = p
1	$K(n, \mathbf{m})$	Prob(T_{ijkl}=1) = p_{ij}
2	$K(n, \mathbf{m}, [I_{ij}])$	All selections of I_{ij} value-pairs from the $m_i \times m_j$ possible pairs are equally likely

Table 4.1 - Summary of Nadel's models.

Nadel develops complexity estimates using the level-1 and level-2 models. His work shows how level-2 is more complex than level-1, but that it is acceptable to use expressions developed for level-1 in place of those for level-2 if we set p_{ij} to be the satisfiabilities of the constraints in the particular instance. More specifically if we set;

$$p_{ij} = I_{ij} / m_i m_j \qquad (4\text{-}4)$$

The advantage of this observation is that the resulting expressions for the expected number of nodes and the expected number of compatibility checks at a given level in the search, which Nadel derives for standard backtracking and for forward checking, are simpler to compute.

Other researchers have also looked at probability models for CSPs. Van Hentenryck (Van Hentenryck 1989) summarises much of the work done by Haralick and Elliott. He presents further evidence, supporting their results, which shows how theory-based complexity equations are sufficient to suggest that forward checking is better than backtracking for large and hard problems.

[1] The term T_{ijkl} is the 0/1 value of entry in the constraint relation matrix between variables *i* and *j* having values *k* and *l* respectively.

Smith (Smith 1994) presents work in which she attempts to estimate the number of solutions for randomly generated binary CSPs. Her model is a modification of that used by Haralick and Elliott whereby not all variables in the problem are required to be constrained.

Dent and Mercer (Dent & Mercer 1996) define a more sophisticated model than that of Haralick and Elliott. Their model also allows individual constraint tightnesses and they incorporate information relevant to the constraint graph topology into their theories. In a similar way (Kwan et al 1997) use a model for randomly generated binary CSPs for predicting the proximity of an instance to the phase transition in solubility (Cheeseman et al 1991).

4.2 Nadel's Complexity Equations

In this section we present the significant parts of Nadel's work which provide enough detail of his theoretical complexity equations for us to make use of them. For more details of the derivation of these equations, we refer the reader to (Nudel 1983a). Throughout the presentation, the following symbols are used;

k	- a search level, ranging from 1 to n, the number of variables in the CSP
A_k	- the set of assigned variables at level k
F_k	- the set of future variables at level k
G_k	- the set of previous[2] variables constrained by the variable at level k; these are in a fixed order
g_{jk}	- the *jth* variable in the set G_k
c(alg)	- the expected number of constraint checks for algorithm *alg*
c(alg, k)	- the expected number of constraint checks for algorithm *alg* at level *k*
n(alg)	- the expected number of nodes expanded for algorithm *alg*
n(alg, k)	- the expected number of nodes expanded for algorithm *alg* at level *k*
$\left\|D_{x_i}\right\|$	- the domain size of the variable at level *i*
p_{ij}	- the *looseness* or *satisfiability* of the constraint between variables *i* and *j*. It is the opposite of tightness and is equal to 1-*p2*

[2] future variables for forward checking

Before giving the detail we outline a set of assumptions which have been made. It is important to be aware of these assumptions so that we can assess the usefulness of the Nadel's work for cases where they might be relaxed.

4.2.1 Working Assumptions

There are four main assumptions which Nadel applied during the development of his models. These working assumptions are;

A_4.1 The models and equations presented are applicable to binary CSPs.

A_4.2 The satisfiability of any constraint C_{xy} is independent of any constraint C_{wz} and independent of the values assigned to variables x and y.

A_4.3 The expressions for the expected numbers of checks and expected number of nodes are for finding *all* solutions.

A_4.4 The level-1 model is an accurate representation of the level-2 model when (4-4) is applied.

These assumptions are the same as the ones used by Haralick and Elliott (Haralick & Elliott 1980) with the exception of A_4.4 since the model used there was only for level-0 classes of problem. Assumption A_4.4 is adopted in line with (Nudel 1983a).

4.2.2 Application to the Standard Backtracking Algorithm

The standard backtracking algorithm (*bt*), also known as chronological backtracking, is one of the simplest systematic algorithms used for solving constraint satisfaction problems. The pseudo code for the algorithm[3] is given in figure 4.1.

[3] This pseudo code is a recursive version taken from (Tsang 1993).

```
PROCEDURE BT(Z, D, C)
BEGIN
   BT_k(Z, {}, D, C);
END

PROCEDURE BT_k(UNLABELLED, COMPOUND_LABEL, D, C)
// UNLABELLED is a set of variables to be labelled
// COMPOUND_LABEL is a set of labels already committed to
BEGIN
   IF(UNLABELLED={}) THEN return(COMPOUND_LABEL);
   ELSE BEGIN
      Pick one variable x from UNLABELLED;
      REPEAT
         Pick one value v from Dx;
         Delete v from Dx;
         IF (COMPOUND_LABEL + {<x,v>} violates no constraints)
         THEN BEGIN
            Result ← BT_k(UNLABELLED-{x}, COMPOUND_LABEL +
                                                {<x,v>}, D, C);
            IF(Result ≠ NIL)THEN return(Result)
         END
      UNTIL (Dx={});
      return(NIL);          // No solution
   END
END
```

Figure 4.1 - The standard backtracking algorithm

In general, for a given search algorithm, *alg*, the total number of nodes visited during search is equal to the sum of nodes expanded at each level, *k*, in the search;

$$n(alg) = \sum_{k=1}^{n} n(alg,k) \tag{4-5}$$

The total number of constraint checks carried out during the search is equal to the sum the expected number of constraint checks performed at each level, *k*, in the search. This is given by ;

$$c(alg) = \sum_{k=1}^{n} c(alg,k) \tag{4-6}$$

These expressions give us the top level calculation required for estimating nodes and constraint checks for specific algorithms, when finding all solutions. For the standard backtracking algorithm in particular, the expected number of nodes at level *k* is effectively the number values that the algorithm attempts to label at that level. This is determined by multiplying the product of the domains of all variables up to the search level *k*, by the probability that all constraints connected between variables at search levels preceding *k* were satisfied. That probability is given in (4-7a).

$$\prod_{i<j\in A_{k-1}} p_{ij} = \prod_{i<A_{k-1}} \prod_{\substack{j<i \\ j\in A_{k-1}}} p_{ij} \tag{4-7a}$$

Using (4-7a), we obtain (Nudel 1983a);

$$n(bt,k) = \left(\prod_{i\in A_k} \left| D_{xi} \right| \right) \left(\prod_{i<j\in A_{k-1}} p_{ij} \right) \tag{4-7}$$

The expected number of checks per node is dependent on how many constraints are connected to the variable at the current search level, *k*. Furthermore, we only continue checking the constraints if all previous checks against the current assignment are successful. The total expected number of constraint checks at level *k* is therefore equal to the sum of the probabilities of each constraint between x_k and previously assigned variables is checked, multiplied by the number of nodes expanded at that level;

$$c(bt,k) = \left(\sum_{i=1}^{|G_k|-1} \prod_{j=1}^{i-1} p_{g_{jk}k} \right) \times n(bt,k) \tag{4-8}$$

Combining (4-6), (4-7) and (4-8), we obtain an estimate for the total number of constraint checks performed by standard backtracking;

$$c(bt) = \sum_{k=1}^{n} c(bt,k) \tag{4-9a}$$

$$c(bt) = \sum_{k=1}^{n} \left(\sum_{i=1}^{|G_k|-1} \prod_{j=1}^{i-1} p_{g_{jk}k} \right) \times \left(\prod_{i\in A_k} \left| D_{xi} \right| \right) \left(\prod_{i<j\in A_{k-1}} p_{ij} \right) \tag{4-9b}$$

4.2.3 Application to the Forward Checking Algorithm

The forward checking algorithm (*fc*) uses the concept of *lookahead* in order to detect futile regions of the search space (Tsang 1997). The increased sophistication associated with lookahead can lead to more effective problem solving and has been used as the basis for some commercial

constraint programming languages (ILOG 1994) (Dincbas et al 1988). The pseudocode for forward checking is given in figure 4.2.

One key difference for the forward checking algorithm, when compared with standard backtracking, is that we have the notion of a survival probability for a future variable *f* at level *k*, where *k* is less than *f*. This is equal to 1 minus the probability of all values in a domain being rejected and is given by;

$$S_f^k = 1 - \left(1 - \prod_{i \in A_k} p_{fi}\right)^{|D_f|} \tag{4-10}$$

where D_f is the domain of future variable *f*. Nadel shows that the number of nodes at level *k* is then obtained by combining the product of the domain sizes of variables up to level *k* by the probability that all constraints on those variables are satisfied and the probability that no future variable incurred a domain wipe out at the previous level. This gives us;

$$n(fc,k) = \left(\prod_{i \in A_k} |D_{x_i}|\right) \times \left(\prod_{i<j \in A_k} p_{ij}\right) \times \left(\prod_{f \in F_k} S_f^{(k-1)}\right) \tag{4-11}$$

```
PROCEDURE FC(Z, D, C)
BEGIN
   FC_k(Z, {}, D, C);
END

PROCEDURE FC_k(UNLABELLED, COMPOUND_LABEL, D, C)
BEGIN
   IF(UNLABELLED={}) THEN return(COMPOUND_LABEL);
   ELSE BEGIN
      Pick one variable x from UNLABELLED;
      REPEAT
         Pick one value v from Dx;
         Delete v from Dx;
         D`← Update(UNLABELLED - {X}, D, C,
                                          COMPOUND_LABEL+{<x,v>});
         IF (no domain in D` is empty)
         THEN BEGIN
            Result ← FC_k(UNLABELLED-{x}, COMPOUND_LABEL +
                                                {<x,v>}, D`, C);
            IF(Result ≠ NIL)THEN return(Result)
         END
      UNTIL (Dx={});
      return(NIL);          // No solution
   END
END

PROCEDURE Update(W, D, C, COMPOUND_LABEL)
BEGIN
   D` ← D;
   FOR each variable y in W DO;
      FOR each value v in Dy` DO;
         IF(<y, v> is incompatible with COMPOUND_LABEL with respect
            to constraints on y +COMPOUND_LABEL)
         THEN Dy`← Dy` - {v};
   return(D`);
END
```

Figure 4.2 - the forward checking algorithm

In order to calculate the number of constraint checks made per node, we first need an expression for the expected reduced domain size of future variable *f* at level *k*. This is dependent on the original domain size of the variable and its compatibility with past variables. Nadel defines it to be;

$$\left|D_k^f\right| = \left|D_f\right| \left(\prod_{j \in A_{k-1}} p_{fj} \right) \Big/ S_f^{(k-1)} \qquad (4\text{-}12)$$

Finally Nadel defines the number of constraint checks performed at level *k* by multiplying the number of nodes at that level by the sum of probabilities that each constraint with future variables is checked. This gives;

$$c(fc,k) = n(fc,k) \times \sum_{i=1}^{|G_k|} \left(\left| D_k^{g_{ik}} \right| \prod_{j=1}^{i-1} S_{g_{jk}}^{k} \Big/ S_{g_{jk}}^{(k-1)} \right) \tag{4-13}$$

Using (4-6), (4-11) and (4-13) we now have a complete expression for estimating the total constraint checks performed by forward checking;

$$c(fc) = \sum_{k=1}^{n} c(fc,k) \tag{4-14a}$$

$$c(fc) = \sum_{k=1}^{n} \left(\sum_{i=1}^{|G_k|} \left(\left| D_k^{g_{ik}} \right| \prod_{j=1}^{i-1} S_{g_{jk}}^{k} \Big/ S_{g_{jk}}^{(k-1)} \right) \right) \times \left(\prod_{i \in A_k} \left| D_{x_i} \right| \right) \left(\prod_{i<j \in A_{k-1}} p_{ij} \right) \left(\prod_{f \in F_k} S_f^{(k-1)} \right) \tag{4-14}$$

4.2.4 The Expected Number of Solutions for Standard Backtracking and Forward Checking

The expected number of solutions to a problem can be viewed as the expected number of nodes remaining at the bottom of the search tree. From equation (4-7) for the standard backtracking algorithm we see that this reduces to;

$$S = \left(\prod_{i \le n} \left| D_{x_i} \right| \right) \left(\prod_{i<j \le n} p_{ij} \right) \tag{4-15}$$

The same result is found if we reduce equation (4-11) for the forward checking algorithm. This is to be expected since the number of solutions to a problem is an algorithm independent property of a CSP. We also note that if the domain size is fixed and p_{ij} is set to p, then (4-15) reduces to (4-1).

4.3 Extensions to Backjumping

The expected complexity work by Nadel was concerned with two algorithms which backtrack chronologically. This means that on detection of a dead end in the search these algorithms backtrack to the previous choice point. This mechanism has a major flaw in that it is prone to the pathological behaviour known as *thrashing*. Thrashing involves the repeated futile search of

regions in the search space which are bound to fail, due to some culprit decision prior to the backtrack point.

A major class of algorithms which alleviates the phenomenon of thrashing, to varying degrees, is that of *intelligent backjumping* algorithms. The idea of intelligent backjumping is to identify the reason, or culprit decision, which caused a backtrack to take place. Of course, this process comes with some overhead and the more sophisticated the culprit identification mechanism, the greater the overhead we can expect. However, it has been shown that the use of intelligent backjumping can guarantee a reduction in the cost of search in terms of the number of constraint checks performed (Kondrak & van Beek 1995). An example of one such algorithm is backjumping. The pseudocode for backjumping is given in figure 4.3.

In the remainder of this section we present a significant extension of Nadel's work. We develop an expression for the expected cost of searching binary CSPs using the backjumping algorithm. This is important as it allows us to make comparisons of the expected search costs of different *ZDC* formulations of problems with respect to that algorithm.

```
PROCEDURE BJ(Z, D, C)
BEGIN
   BJ_k(Z, {}, D, C, 1);
END

PROCEDURE BJ_k(UNLABELLED, COMPOUND_LABEL, D, C, L)
BEGIN
   IF(UNLABELLED={}) THEN return(COMPOUND_LABEL);
   ELSE BEGIN
      Pick one variable x from UNLABELLED;
      Level_of[x] ← L; TDx ← Dx;
      REPEAT
         v ← any value from TDx;
         TDx ← TDx - {v};
         IF (COMPOUND_LABEL + {<x,v>} violates no constraints)
         THEN BEGIN
            Result ← BT_k(UNLABELLED-{x}, COMPOUND_LABEL +
                                                {<x,v>}, D, C, L+1);
            IF(Result ≠ backtrack_to(Level))
            THEN return(Result);
         END
      UNTIL ((Tdx={}) OR (Result=backtrack_to(Level) AND Level<L));
      IF(Result=backtrack_to(Level) AND Level < L)
      THEN return(backtrack_to(Level));
      ELSE BEGIN
         Level ← Analyse_bt_level(x, COMPOUND_LABEL, Dx, C, L);
         return(backtrack_to(Level));          // No solution
      END
   END
END

PROCEDURE Analyse_bt_level(x, Compound_label, Dx, C, L)
BEGIN
   Level ← -1;
   FOR each (a∈ Dx) DO
   BEGIN
      Temp ← L-1; NoConflict ← True;
      FOR each <y,b> ∈ Compound_label DO
         IF NOT satisfies((<x,a><y,b>), Cx,y)
         THEN BEGIN
            Temp ← Min(Temp, Level_of[y]);
            NoConflict ← False;
         END
      IF(NoConflict) THEN return(Level_of[x] - 1);
   END
   return(Level);
END
```

Figure 4.3 - the backjumping algorithm.

As a working example and to aid the clarity of our presentation, we shall refer to the CSP in figure 4.4.

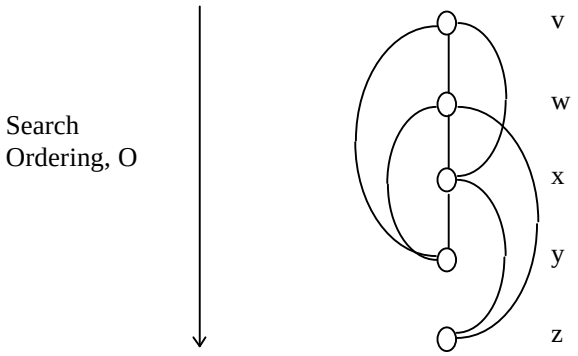

Figure 4.4 - An example CSP

4.3.1 The Effects of Backjumping

As we have mentioned, one of the problems with standard backtracking is that it suffers from the pathological behaviour known as thrashing. In our example CSP, consider the scenario where we have a partial assignment over the variables *v*, *w*, *x* and *y* such that there is no compatible value which can be assigned to *z*. Standard backtracking will progress and attempt to assign a value to variable *z*. This will fail for each of the values in the domain of *z* and once all candidates are exhausted, a backtrack is made such that a new value is assigned to *y*. The algorithm then repeats its attempt to assign a value to *z*. Clearly this will not succeed since *z* is not constrained by *y*. As a result, futile attempts to assign a value to *z* are continually repeated until the algorithm eventually backtracks far enough so that the reason for its incompatibility is changed.

Backjumping can reduce the level of thrashing by noting the first order reasons for failure. Referring again to our example, when it is found that no value exists for variable *z*, which is compatible with the previous assignments, the algorithm jumps back to either variable *x* or variable *w* depending on the exact reason for failure. This simple improvement can result in significant savings in the overall search costs.

4.3.2 Extending Theoretical Complexity Estimates to Backjumping

The key difference between backtracking and backjumping is the ability to detect a reason for failure when no compatible value can be found for a particular variable. In doing this, the backjumping algorithm reduces the number of times a particular sub-search space is needlessly expanded. We can see this effect more clearly if we again consider our example CSP in figure 4.4.

Taking a constraint in isolation, C_{vy}, say, if we ignore the effect of all other constraints then any backjumping which takes place from *y* to *v* will result in a reduction in the number of times values from the domains of variables *w* and *x* are expanded. This is so because the jump back means any remaining values in their domains are skipped. We can view this process as reducing the *effective domain size* of those affected variables, and this manifests itself through a reduction in the number of nodes expanded, relative to those expanded by standard backtracking. We indicate the effective domain size of variable x_i with $eds(x_i)$.

Our first aim in terms of estimating the expected complexity of backjumping is to determine an expression for the expected number of nodes at each level *k* in the search, thus allowing us to use (4-5) in order to calculate the total number of nodes explored. The number of nodes at level *k* can be seen as the set of possible values for the variable at that level, ready to be expanded further to the next level. It can be expressed in a similar way to that of standard backtracking (4-7) but this time we include the notion of effective domain size of variable x_i, $eds(x_i)$;

$$n(bj,k) = \left(\prod_{i \le k} eds(x_i)\right) \times \left(\prod_{i < j \in A_{k-1}} p_{ij}\right) \qquad (4\text{-}16)$$

There are several factors which need to be taken into account in order to evaluate the effective domain size of a variable. To help clarify our reasoning, consider the single constraint C_{wy} in figure 4.4, while ignoring all other constraints in the problem. Backjumps arising from this constraint have the potential of affecting the effective domain size of variable *x* - i.e. the variable spanned by the constraint. There are two factors which affect this effective domain size. These are;

- *The current search level* - if the current search level is at variable *x* or variable *y*, then the constraint C_{wy} can reduce $eds(x)$. However, for search levels below variable *y*, this constraint only has the same effect as with standard backtracking since only labels consistent with that constraint can be extended. Since it is clearly a function of the current search level, we modify our notation for the effective domain size of variable x_i, given a current search level *k*, to be $eds(x_i, k)$, where $i<k$.
- *The probability of jumping back across the variable* - if we reach level *k* in the search, then this probability is the net effect of all possible jumps originating at variable x_k and all future variables. In our example scenario, if the current search level is *x*, then if a compatible value is found for *x*, we could proceed to *y* and then jump back over level *x*. When considering the nodes expanded at level *y*, the potential exists for a jump to occur when all the values of *y* fail. We denote this probability of jumping across a particular level *i*, given a current search level *k*, from future level *j* to be $p(jump(i, j, k))$, where $i<k\leq j$. When there are constraints at many future levels, we must consider the net effect of all their probabilities. Given a current search level *k*, the net probability of jumping across a particular level *i*, from x_k or its future variable, is denoted as $p(jump(i, k))$.

Taking the above factors into account, we can now give an initial expression for the effective domain size of a variable, given current search level *k*;

$$eds(x_i,k) = 1+\left(\left(\left|D_{x_i}\right|-1\right)\times\left(1-p(\,jump(i,k))\right)\right) \tag{4-17}$$

Equation (4-17) says that the effective domain size of a variable, given current search level *k*, is equal to 1, an initial value, plus the remainder of the domain, which is expanded with the probability that no jump occurred back across the first value. We also notice that the value of *p(jump(i, k))* is the same for all members of the domain of x_i since it is determined by assignments to variables in the search previous to x_i.

To obtain the value of *p(jump(i, k))*, we need to consider all of the possible jumps from levels at $j \geq k$. We can say that *p(jump(i, k))* is equal to 1 minus the probability of no jump occurring from levels *k* of later. This is given by;

$$p(\,jump(i,k)) = 1-\left(\prod_{j\geq k}(1-p(\,jump(i,j,k)))\right) \tag{4-18}$$

The final part of our expression for $eds(x_i, k)$ is to develop an expression for *p(jump(i,j,k))*. This probability comprises two parts. The first is the probability of actually getting to level *j*, given that we have already reached level *k* and we denote this to be *p(at(j, k))*. In order to reach level *j* there must be at least one compatible value at each of the levels between *i* and *j*. This only occurs if all constraints between these variables and past assignments are satisfied. *p(at(j, k))* is therefore equal to the grand product of the constraints between those variables and other past variables;

$$p(at(\,j,k)) = \prod_{\substack{k\leq l<j \\ 1<m<l}} p_{l,m} \tag{4-19}$$

The second part of *p(jump(i,j,k))* is the probability of a jump occurring from level *j* to a level above level *i*. We call the probability of these jumps *p(jump_at(j, i))*. They occur when all of the values in the domain of x_j are incompatible with a variable at levels higher than *i*;

$$p(\,jump\_at(\,j,i)) = \left(1-\prod_{l<i} p_{lj}\right)^{\left|D_{xj}\right|} \tag{4-20}$$

Combining (4-19) and (4-20) we get the overall expression for *p(jump(i,j,k))*;

$$p(\,jump(i,j,k)) = \prod_{\substack{k\le l<j \\ 1<m<l}} p_{l,m} \times \left(1-\prod_{l<i} p_{lj}\right)^{\left|D_{xj}\right|} \tag{4-21}$$

Equations (4-16) to (4-21) give us all the components we need in order to calculate *n(bj, k)*. Notice that if all the *p(jump(i,j,k))* values are equal to 0 then *eds(x_i, k)* becomes equivalent to $\left|D_{xi}\right|$ and hence the value of *n(bj, k)* becomes equal to *n(bt, k)*.

In terms of constraint checks per node at a level *k*, the method of calculation is the same for both backtracking and backjumping since the same constraint checks have to be made. We can therefore use (4-8) for the purposes of backjumping.

$$c(bj,k) = \sum_{i=1}^{|G_k|-1} \prod_{j=1}^{i-1} p_{g_{jk}k} \times n(bj,k) \tag{4-22}$$

Combining equations (4-6) and equations (4-16) and (4-22) we have our expression for the total number of constraint checks performed by backjumping;

$$c(bj) = \sum_{k=1}^{n}\left(\sum_{i=1}^{|G_k|-1} \prod_{j=1}^{i-1} p_{g_{jk}k}\right) \times \left(\prod_{i\le k} eds(x_i)\right) \times \left(\prod_{i<j\in A_{k-1}} p_{ij}\right) \tag{4-23}$$

One further important consideration is the expected number of solutions to the problem. This is equal to the number successful nodes at level *n*, or the number of nodes at level *n+1*. Since no jumping can occur across this level, *p(jump(i,j,k))* is reduced to 0 for all *j*, and hence the effective domain size is equal to the actual domain size for all variables. The result of this is that (4-16) reduces to (4-15), hence the expected number of solutions is the same for backjumping as for standard backtracking and forward checking. This is important because we should expect this property to be algorithm independent.

4.4 The Accuracy Of Theoretical Estimates

So far we have only described the theoretical aspects of Nadel's work. These provide us with a tool for making estimates of search costs. We now need to consider the accuracy of this approach. No work has been published which provides any evaluation of the usefulness of the theoretical complexity estimates other than the work of Nadel. In the remainder of this chapter we present an important assessment of the approach.

Nadel makes an observation about the observed accuracy of his equations, stating he had found that;

Observation 4.2 (Nadel 1990b): About 85% of the instances within a problem class have *exact-case* complexity of solution within 15% of the *expected* class average.

In this section we look to assess this observation and consider two aspects of the accuracy of the theory-based estimates given in sections 4.2 and 4.3

1. The accuracy of the complexity models on a class of problems.
2. The general accuracy of the complexity models.

The first of these is important because it dictates our chances of making a prediction which is approaching the mean for that class of problem. The second is important because it impacts on the possibility of using the approach more widely.

4.4.1 Accuracy Within a Problem Class

4.4.1.1 Motivation

Nadel's level-2 model applies to classes of problems defined by the tuple $K(n, \boldsymbol{M}, [I_{ij}])$. He refers to this as a *constraint-* or *c-class* of problems because it is defined to the level of the actual detail of the constraints. Clearly there is likely to be some degree of variance in the search costs within a given class of problems and as a result this will affect the proximity of any expected complexity value, which is applicable to the problem class, and the actual cost of solving a particular instance within that class. The essence of observation 4.2 is that the major proportion of instances are reasonably close to the expected complexity value.

For a given c-class, the number of possible instances that can exist within that class definition is dependent on the number of different ways of choosing the I_{ij} compound labels of each constraint from the $m_i \times m_j$ possible compound labels between the pairs of variables. This is given by;

$$|K(n, \boldsymbol{m}, [I_{ij}])| = \prod_{i<j} \binom{m_i \times m_j}{I_{ij}} \tag{4-24}$$

This allows for a significant range of possible instances. As an illustration of how instances of the same c-class might vary, let us consider the CSPs in figure 4.5. There, three of the possible CSP instances from the class $K(3, [3, 3, 3], [5/9, 4/9])$ are shown. Equation (4-15) gives us the expected number of solutions for this class of problems to be 6.6. However if we look at the actual number of solutions, there are 7 for scenario 1, 2 for scenario 2 and 12 for scenario 3. Of course this is only a very small sample of the full range of possibilities for this particular c-class. However, it serves to demonstrate how the range of values for a given c-class of CSPs can vary, and this can be expected to apply to the number of solutions, nodes expanded during search and the number of compatibility checks performed.

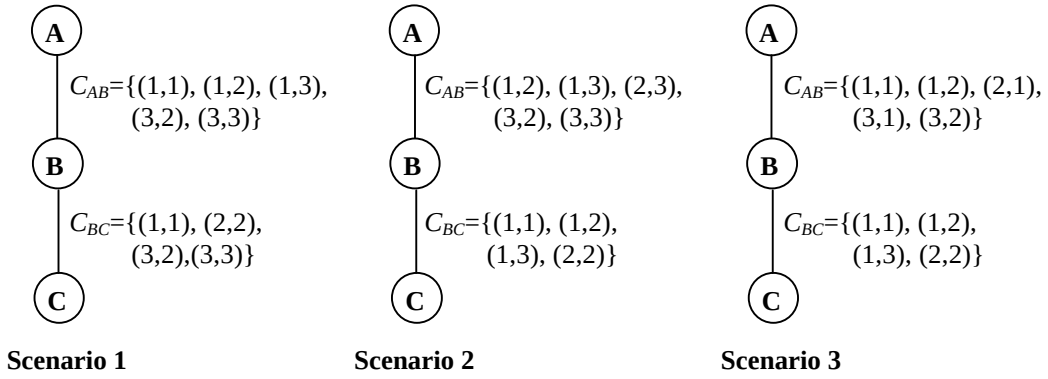

Figure 4.5 - Instances of the same c-class *K(3, [3, 3, 3], [5/9, 4/9])*

If the variance in complexity values for a c-class is high then we may have to question the reliability of any estimate, when applying it to a particular instance of that c-class, since the probability of a particular instance being near to the predicted c-class average is reduced. The basis of Nadel's observation was empirical data generated using a restricted selection of very small problems. We believe that further evidence was required in order to investigate the accuracy of theoretical complexity estimates more widely. In order to achieve this we carried out a series of experiments on randomly generated binary CSPs.

4.4.1.2 Experimental Details

Our approach was to randomly generate sets of binary CSPs defined by the tuple <*n, m, p1, p2*> where *n* is the number of variables, *m* is the uniform domain size, *p1* is the density of the constraint graph and *p2* is the tightness of the constraints as defined in chapter 2. For each given class of problems defined by this tuple we generated 100 instances with different random constraint matrices[4]. Our aim was to look at the variation in actual search costs over a range of constraint graph densities and for different problem sizes. These were then compared with the actual complexity estimate of the classes.

For the problem class <10, 10, 1.0, 0.4> we ran the standard backtracking, backjumping and forward checking algorithms. The natural lexical variable ordering (*nat*) was used. Figures 4.6-4.8 show the distributions of the actual search costs for the 100 instances. Each of these figures also has an indication of the location of the expected c-class average.

[4] Full details of our problem generator are given in appendix A1

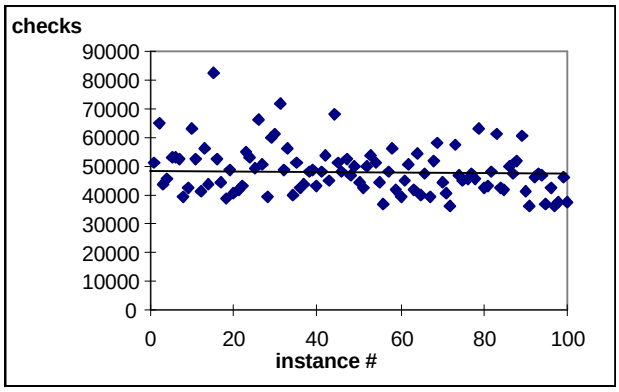

Figure 4.6 -Performance of bt on problem class <10, 10, 1.0, 0.4>

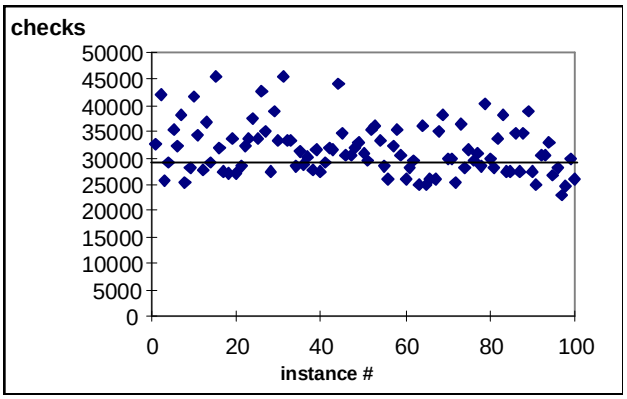

Figure 4.7 -Performance of bj on problem class <10, 10, 1.0, 0.4>

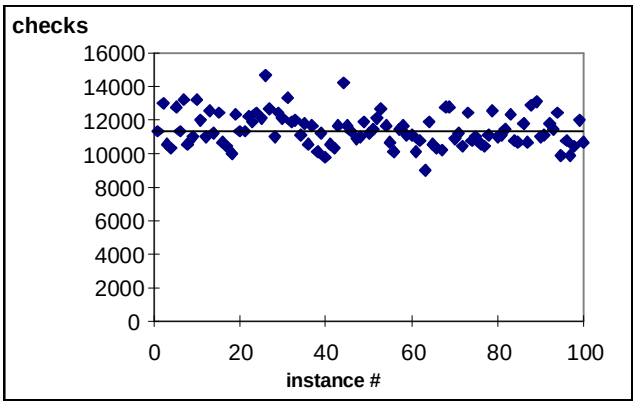

Figure 4.8 -Performance of fc on problem class <10, 10, 1.0, 0.4>

A similar set of experiments was carried out with problems classes having 20 variables. For these problems we only consider the forward checking and backjumping algorithms since the cost in finding all solutions on these larger problems with standard backtracking can become prohibitive. Table 4.2 shows our results for the experiments with both 10 and 20 variables.

In table 4.2, the third and fourth columns show the actual observed mean cost of solving the instances and the expected class average, as given by the theoretical complexity estimates. In the fifth and sixth columns, we calculate the percentage of individual instances that show an actual search cost within 15% of the measured mean, or the expected class average. This gives us a direct comparison with observation 4.2.

Problem Class	Algorithm	Mean constraint checks	Expected c-class average	% instances within 15% of mean	% instances within 15% of expected c-class average
<10,10,0.2,0.88>	bt	27465	28871	72	67
<10,10,0.5,0.63>	bt	48968	48402	71	68
<10,10,1.0,0.40>	bt	48459	47689	67	70
<10,10,0.2,0.88>	bj	9988	9546	58	57
<10,10,0.5,0.63>	bj	13887	19362	44	20
<10,10,1.0,0.40>	bj	31594	28807	71	65
<20,10,0.2,0.63>	bj	2.7×10^8	4.1×10^8	35	15
<20,10,0.5,0.38>	bj	3.0×10^7	3.7×10^7	60	28
<20,10,1.0,0.21>	bj	2.3×10^7	2.3×10^7	74	74
<10,10,0.2,0.88>	fc	2732	2888	51	45
<10,10,0.5,0.63>	fc	2426	2434	79	79
<10,10,1.0,0.40>	fc	11433	11122	94	90
<20,10,0.2,0.63>	fc	8.8×10^6	9.0×10^6	26	29
<20,10,0.5,0.38>	fc	1.0×10^6	995565	65	65
<20,10,1.0,0.21>	fc	3.1×10^6	3.0×10^6	87	79

Table 4.2 - Estimated and actual means for selected class of problem, using the natural lexical ordering for each algorithm. 100 instances were generated per c-class.

4.4.1.3 Analysis of Results

Our results in figures 4.6-4.8 and in table 4.2 show that there is indeed a distribution of complexities within a c-class which has a concentration of instances having values close to the c-class average. This is particularly the case for the problem classes which have fully connected, high density constraint graphs, such as classes <20, 10, 1.0, 0.21> and <10, 10, 1.0, 0.40>.

The aim of the experiments in this section was to investigate Nadel's observation. It is clear from our results that with the problem classes used here, observation 4.2 does not hold in general, especially for CSPs with lower density constraint graphs. The ramification of this observation is that if we are given a single CSP instance and wish to use the complexity equations for estimating the expected cost of solving a particular instance, we should expect relatively poor accuracy for some classes, especially for those classes with lower density constraint graphs.

One other observation from our results is that our estimate for backjumping performs with an accuracy comparable with the estimates for forward checking and standard backtracking. However, it should be noted that the spread of actual costs within a given c-class appear to be greater for backjumping than the other algorithms. This suggests that, especially for CSPs with lower density constraint graphs, the accuracy of c-class estimates using (4-23) is likely to be reduced.

4.4.1.4 Conclusions

The observations above, particularly with reference to the accuracy of our estimates on low density CSPs, are similar to results in previous work where probabilistic models of CSPs were found to be less accurate as the constraint graph density becomes very low (Kwan et al 1996) (Dent & Mercer 1996). However, this still leaves a significant group of problem classes where equations (4-9), (4-14) and (4-23) are reasonably accurate. In the next section we consider the general accuracy of these equations when used to estimate the complexity of single problem instances.

The results in table 4.2 also show the importance of not rejecting other properties of ZDC formulations when developing evaluation heuristics as we conjectured in chapter 3. In particular, they suggest that expected complexity measures in conjunction with constraint density are strong candidates for playing a significant part of such heuristics.

4.4.2 General Estimation Accuracy

We have seen in the previous section how the complexity over a given c-class can vary. In this section we look at the general accuracy of Nadel's estimations for standard backtracking and for forward checking, as well as for our new expression for backjumping. Our approach is to consider

three different types of problem; randomly generated binary CSPs, graph colouring problems and the *n*-Queens problem.

4.4.2.1 Accuracy with Randomly Generated Binary CSPs

Since equations (4-9), (4-14) and (4-23) are derived from a probabilistic model, randomly generated CSPs are a good candidate for assessing their underlying accuracy - if they are not accurate for random problems, then it is unlikely that they will be accurate for non-random ones. For our experiments we used a range of binary random CSPs defined by the tuple *<n, m, p1, p2>*, as in section 4.4.1 in order to do this. However, for this particular experiment we used a different c-class for each instance - i.e. a different constraint graph was used for each one.

The sample size for each set of parameters used was 100, and for each instance in a given set all three algorithms were run using the minimum width variable ordering heuristic (Freuder 1982). The heuristic was used in order to reduce the amount of time taken to solve the CSPs[5]. The observed cost for solving each instance was recorded and then compared with the estimated number of constraint checks.

In order to give a concise overview of our results we analysed each set with respect to four parameters;

i. minimum ratio of expected and measured search cost
ii. maximum ratio of expected and measured search cost
iii. mean ratio of expected and measured search cost
iv. standard deviation of ratio of expected and measured search cost

The above parameters give a good indication of the worst case accuracy of our estimate, together with a typical accuracy. The complete set of results are given in table 4.3.

[5] This is acceptable since our complexity equations can be used with any static variable ordering.

		Expected Cost / Measured Cost			
Algorithm	Problem Class	Minimum	Maximum	Mean	Standard Deviation
bt	<10,10,0.2,0.88>	0.31	3.59	1.42	0.77
bj	<10,10,0.2,0.88>	0.22	4.94	1.30	0.84
fc	<10,10,0.2,0.88>	0.29	3.84	1.20	0.60
bt	<10,10,0.5,0.63>	0.49	1.48	1.02	0.20
bj	<10,10,0.5,0.63>	0.43	1.50	0.95	0.21
fc	<10,10,0.5,0.63>	0.66	1.68	1.00	0.16
bt	<10,10,1.0,0.40>	0.70	1.52	1.02	0.15
bj	<10,10,1.0,0.40>	0.64	1.43	0.95	0.13
fc	<10,10,1.0,0.40>	0.81	1.22	0.99	0.08
bt	<20,10,0.2,0.63>	0.24	17.00	2.1	2.28
bj	<20,10,0.2,0.63>	0.27	17.97	2.9	3.14
fc	<20,10,0.2,0.63>	0.26	4.28	1.35	0.84
bt	<20,10,0.5,0.38>	0.62	1.90	1.06	0.24
bj	<20,10,0.5,0.38>	0.62	2.04	1.16	0.26
fc	<20,10,0.5,0.38>	0.71	1.44	0.99	0.16
bt	<20,10,1.0,0.21>	0.57	1.50	1.03	0.18
bj	<20,10,1.0,0.21>	0.51	1.31	0.88	0.14
fc	<20,10,1.0,0.21>	0.63	1.16	0.95	0.11
bt	<30,10,0.2,0.52>	0.42	5.67	1.27	0.67
bj	<30,10,0.2,0.52>	0.58	7.70	1.75	0.86
fc	<30,10,0.2,0.52>	0.47	2.60	1.10	0.36
bt	<30,10,0.5,0.26>	0.49	1.72	1.12	0.28
bj	<30,10,0.5,0.26>	0.58	1.96	1.30	0.31
fc	<30,10,0.5,0.26>	0.59	1.49	1.00	0.18

Table 4.3 - General accuracy of estimated complexities for bt, bj and fc with randomly generated binary CSPs. A sample size of 100 problems per class was used.

Our results are significant in that they show a reasonable degree of accuracy for all of the expected complexity expressions, except for those with the lower density problems. However, only the two sets of results, which are shaded grey, showed an average worse than a factor 2 error. This is in line with the results obtained in section 4.4.1. Most estimates of the worst case search costs are well within a factor 2 of the observed cost, for all of the algorithms. The average ratio of estimated to measured is much closer to 1.0 in most cases.

4.4.2.1.1 Conclusions

These results, combined with those in section 4.4.1, suggest that we can use the complexity estimates reliably for randomly generated binary CSPs with high density constraint graphs. Furthermore, our estimate for backjumping is seen to perform in line with those for standard backtracking and forward checking. This again gives us a high degree of confidence in its correctness.

4.4.2.2 Accuracy in Graph Colouring Problems

The second type of problem that we used to investigate the accuracy of the expected complexity equations was graph-colouring problems. These were chosen because they differ significantly form the previous sets of problems in that the constraints are all exactly the same - i.e. they are all "not equals" constraints. This reduces the level of randomness in the problem structure which is implicit in the probabilistic nature of Nadel's models. As a result we can expect the level of accuracy of the theoretical complexity estimates to be reduced. If the estimates are generally applicable, then we should expect them to be reasonably accurate despite this reduction in randomness.

In order to test this conjecture we used randomly generated graph colouring problems based on the tuple <*n, m, c*> where *n* is the number of nodes in the graph, *m* is the number of colours used and *c* is the average connectivity of each node in the graph. Two combinations of these parameters were used with a sample of 100 problems for each. As in section 4.4.2.1, all three of our algorithms, combined with the minimum width variable ordering heuristic, were used to solve the instances for finding all solutions. Our results were processed in the same way, as shown in table 4.4.

		Expected Cost / Measured Cost			
Algorithm	Problem Class	Minimum	Maximum	Mean	Standard Deviation
bt	<50,3,4.5>	0.32	3184.91	191.43	529.01
bj	<50,3,4.5>	0.05	477.45	37.22	96.15
fc	<50,3,4.5>	0.08	141.94	8.33	23.23
bt	<25,5,9.6>	6.23	9787.54	589.95	1740.59
bj	<25,5,9.6>	2.47	3602.63	233.26	631.62
fc	<25,5,9.6>	2.97	474.58	48.06	75.35

Table 4.4 - General accuracy of estimated complexities for bt, bj and fc with graph colouring problems. A sample size of 100 problems per class was used.

4.4.2.2.1 Conclusions

Clearly, table 4.4 shows that our complexity expressions are not accurate for these types of problem. For the average case, the level of inaccuracy is in excess of an order of magnitude in most cases. There are two possible reasons for this. The first is that these problems have

extremely sparse constraint graphs. Relatively poor performance was also seen for such classes in the previous section, although not on the same scale as seen here. The second possibility is that the reduction in randomness in the structure of the constraints has affected the validity of the estimates.

4.4.2.3 Accuracy with the *n*-Queens Problem

The third group of problems we used to assess the general accuracy of the theoretical complexity estimates was the *n*-Queens problem. The reasons for using this problem are twofold;

1. To extend the experiments by Nadel (Nadel 1990a) which looked at estimates for the *n*-Queens problem, but only up to n=5.
2. The *n*-Queens problem is another example of one having non-random, structured, constraints. This allows us to further investigate the effect of reducing the level of randomness in problems, this time with a fully connected graph.

The *8*-Queens problem is described in detail in chapter 2. For *n*-Queens, we parameterise the number of queens to be placed. The problem is therefore to place n queens on an $n \times n$ chess board. The formulation we used to represent the problem for this experiment was analogous to *8Q_1*;

nQ_1: Z: n variables, v_1, v_2 ... v_n, representing the eight rows on the chess board

D: {A, B, C,...} for each variable, representing the position of the queen in the row - e.g. for *8*-Queens the domain would be {A, B, C, D, E, F, G, H}

C: no queen may attack any other

We ran the same three algorithms on problems for n=3 to n=12 and compared the results with the expected complexity values for the c-classes to which the *n*-Queens instances belong. The variable ordering used was the natural, lexical variable ordering (nat). Our results are given in figure 4.9 - 4.11.

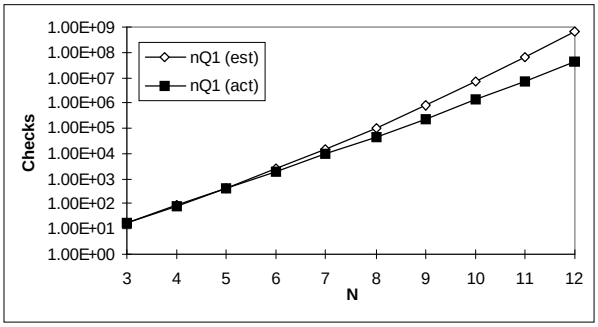

Figure 4.9 - Accuracy of estimation for standard backtracking on *n*-Queens problems

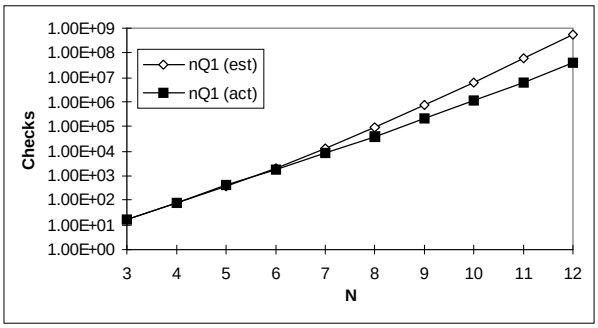

Figure 4.10 - Accuracy of estimation for backjumping on *n*-Queens problems

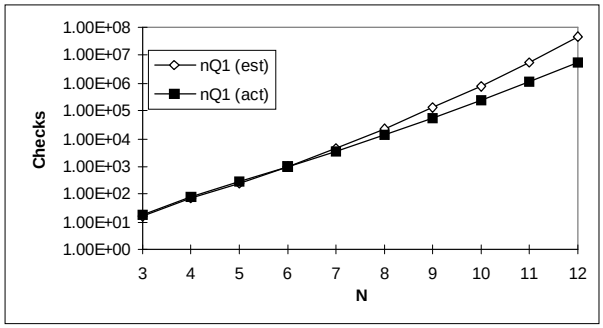

Figure 4.11 - Accuracy of estimation for forward checking on *n*-Queens problems

4.4.2.3.1 Conclusions

The problems used for this experiment violate the random nature of the probability model assumed for the complexity estimates. However, despite this violation, the results suggest reasonable

accuracy when predicting the complexity for our three algorithms, especially for the lower values of *n* where the estimates are consistently within an order of magnitude of the actual observed value.

As *n* increases, it appears that the accuracy of the estimates does degrade. One possible reason for this is that the constraints for *n*-Queens become extremely loose as *n* grows large. As a result, there is greater potential for large groups of legal compound labels to occur. This may exaggerate any adverse effects of reducing the level of randomness implied in the probability models.

4.4.3 Conclusions on the Accuracy of Theoretical Estimates for Predicting Search Costs

From our experiments in this section we have gained more insight into the usefulness of theoretical complexity estimate for estimating search costs of three algorithms. The main findings from these experiments were;

1. The accuracy of the estimates for predicting instance search costs within a problem class is not generally as good as that suggested in observation 4.2. However, we do see that there is a clustering of instance costs about the predicted complexity for the class. We found this clustering to be greatest for problem classes with higher density constraint graphs.
2. The general accuracy of the theoretical complexity estimates is good for randomly generated binary CSPs, especially for those with higher density constraint graphs.
3. Reductions in the level of randomness in the structure of constraints in problems appears to reduce the accuracy of the estimates.

These findings show promise, indicating that the use of theoretical complexity estimates is possible for the purposes of estimating the search cost of algorithm for some problem classes. As a result they have the potential for playing a significant role in the design of *ZDC* formulation evaluation heuristics. In the next section we investigate the direct usefulness of this approach when comparing different *ZDC* formulations of a given problem.

4.5 Applying Theoretical Estimates as *ZDC* Formulation Evaluation Heuristics

In the previous section we considered the use of theoretical complexity estimates for predicting the search costs of solving CSP instances, with respect to their solving by a particular algorithm. One important application of such estimates is with the comparison of different *ZDC* formulations of a problem, as suggested in (Nudel 1983a) (Nadel 1990a). If the theoretical complexity estimates give a good indication of the actual cost of solving a given CSP, then they provide us with a candidate for use as an evaluation heuristic.

Furthermore, the purpose of an evaluation heuristic is to identify the relative expected search cost of a given pair of formulations. This means that even if the estimates are not particularly accurate in terms of predicting the cost of particular instances, they may still be useful, provided they accurately reflect the relative cost. We should remember that what we are looking for is evaluation heuristics which give us this relationship. The important point is the qualitative relationship and not necessarily the exact quantitative nature of it. This is analogous to Nadel's use of estimates to obtain "optimal" search orderings (Nudel 1983a). His results there showed that the estimates accurately reflected the qualitative nature of varying the search ordering.

In the remainder of this section we investigate the application of our theoretical complexity measures to *ZDC* formulation evaluation heuristics. As with previous sections, we consider three types of problem; randomly generated binary CSPs, colouring problems and the *n*-Queens problem.

4.5.1 Randomly Generated Binary CSPs

The first group of problems that we used for the purposes of assessing the usefulness of equations (4-9), (4-14) and (4-23), as the basis for an evaluation heuristic, was that of randomly generated binary CSPs. We required two different formulations of each problem instance considered before any comparison could be made. As the first *ZDC* formulation we used CSPs generated in the same way as we did in section 4.3, using the 4-tuple $<n, m, p1, p2>$. We denote this formulation to be *R1*;

R1 - *Z*: The *n* variables

D: {1..*m*} for each variable.

C: $p1 \times n(n\text{-}1)/2$ random binary matrix constraints each having tightness *p2*.

Our second formulation was based on the idea of taking an *R1* instance and transforming it by replacing pairs of constrained variables with a new variable which is generated by merging the original pair. For example, given a pair of variables x_a and x_b which are constrained by C_{ab}, the domain of the new merged variable, x_{ab}, is given as the set of legal tuples in the constraint C_{ab}. An example of this process is given in figure 4.12.

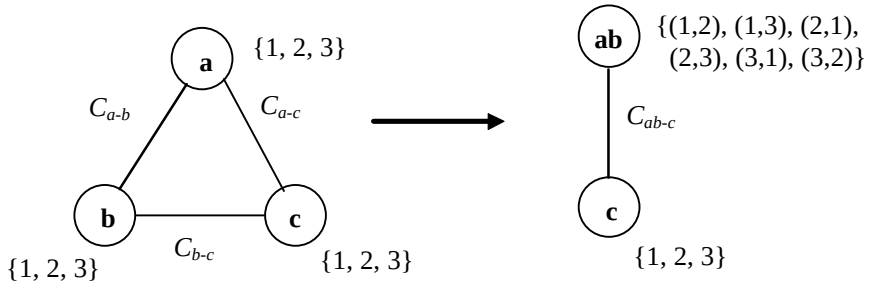

Figure 4.12 - An example of variable merging for the case where the constraint $C_{a\text{-}b}$ is "not equal".

In order to create an alternative *ZDC* formulation we arrange the variables of *R1* in the search ordering we propose to use for *R1*. We then systematically take a single pass through that ordering and merge consecutive pairs of variables which are constrained by each other. Full details of this aggregation transformation we used are given in chapter 5. The result of the transformation process is a new *ZDC* formulation of the problem which we call *R2*;

R2: *Z*: A mixture of merged and non-merged variables. If variables v_i and v_j from *R1* are merged, then they form a single variable $v_{i,j}$ in *R2*, as indicated in figure 4.12.

D: Either {1...*m*} for non-aggregated variables or the set of legal tuples given by an aggregation constraint

C: A combination of original constraints and constraint which have been modified to accommodate the aggregated variables.

For our experiments we used samples of 100 problems for each *n, m, p1, p2* class. We looked at the relative merits of the two candidate *ZDC* formulations, *R1* and *R2*, for each algorithm. We also used the minimum width variable ordering for the larger problems. For these cases, as we described above, we pass through the variables in the search ordering used when solving *R1* when we decide which pairs of variable are to be merged for *ZDC* formulation *R2*. This ensures that we effectively visit the variables in the same order for both formulations[6]. The important point is that we have two *ZDC* formulations which we can expect to have differing search costs.

The results of our experiments are shown in table 4.5. For each instance in a given problem class we compare the expected complexity values of *R1* and *R2* which we call *ec1* and *ec2* respectively. If we find these expected costs to be within a specified percentage of each other, called *margin*, then we regard the two formulations as having the same expected cost in solving and we make no prediction as to which is likely to be the best *ZDC* formulation of the two. Such instances contribute to the "*% no prediction*" column in table 4.5. For cases where the difference is predicted to be greater than the specified percentage, we make a prediction of the actual instance search costs based on the estimates *ec1* and *ec2*. If the actual measured values show the same qualitative relationship as the predictions, we regard it as being a correct prediction. Otherwise, it is regarded as being incorrect.

4.5.1.1 Conclusions

As we can see from the data in table 4.5, equations (4-9), (4-14) and (4-23) were seen to be effective in selecting between *ZDC* formulations *R1* and *R2*. We can say this since the number of times the selections were correct or did not make a prediction was greater than 50% for all problem sets, which was the base criterion laid out in chapter 3. In other words selections based on our theoretical complexity estimates give better results than simply picking one of the two formulations at random. In fact our heuristics performed considerably better than this.

A further, important observation that we can make from table 4.5 is that our heuristics are also effective on problems with low density constraint graphs. This is important because we have previously noted that our complexity equations are less accurate in estimating the actual search cost of algorithms for lower density problems. Our results suggest that when we use different

[6] Issues relating to how variable ordering heuristics should be applied are discussed in detail in Chapter 5.

formulations of a problem, the qualitative nature of the relative complexity values is reflected more reliably.

		Heuristic Accuracy - margin=5%			Heuristic Accuracy - margin=15%		
Algorithm + Heuristic	Problem Class	%correct	%incorrect	% no prediction	%correct	%incorrect	% no prediction
bt+nat	<10,10,0.2,0.88>	73	10	17	60	6	34
bj+nat	<10,10,0.2,0.88>	66	17	17	60	15	25
fc+nat	<10,10,0.2,0.88>	69	11	20	64	7	29
bt+nat	<10,10,0.5,0.63>	77	7	16	62	2	36
bj+nat	<10,10,0.5,0.63>	69	14	17	58	6	36
fc+nat	<10,10,0.5,0.63>	85	5	10	80	2	18
bt+nat	<10,10,1.0,0.40>	100	0	0	100	0	99
bj+nat	<10,10,1.0,0.40>	100	0	0	100	0	0
fc+nat	<10,10,1.0,0.40>	100	0	0	100	0	0
bt+mwo	<20,10,0.1,0.88>	64	29	5	60	27	13
bj+mwo	<20,10,0.1,0.88>	64	30	6	61	23	16
fc+mwo	<20,10,0.1,0.88>	72	23	5	64	19	17
bt+mwo	<10,10,0.2,0.63>	62	23	15	58	19	23
bj+mwo	<10,10,0.2,0.63>	80	16	4	71	8	21
fc+mwo	<10,10,0.2,0.63>	84	13	3	73	10	17
bt+mwo	<10,10,0.5,0.38>	100	0	0	100	0	0
bj+mwo	<10,10,0.5,0.38>	100	0	0	100	0	0
fc+mwo	<10,10,0.5,0.38>	100	0	0	100	0	0
bt+mwo	<30,10,0.07,0.85>	98	2	0	95	2	3
bj+mwo	<30,10,0.07,0.85>	98	1	1	98	1	1
fc+mwo	<30,10,0.07,0.85>	100	0	0	100	0	0
bt+mwo	<30,10,0.2,0.52>	72	22	6	66	19	15
bj+mwo	<30,10,0.2,0.52>	71	26	3	64	23	13
fc+mwo	<30,10,0.2,0.52>	78	19	3	77	19	4

Table 4.5- summary of formulation comparison for randomly generated binary CSPs. 100 problem instances generated per problem class.

4.5.2 Graph Colouring Problems

The second formulation comparison we carried out was with 3- and 5-colouring problems. The aim here was to observe the effect of having all the constraints exactly the same, in this case the "not equal" constraint. As with the previous experiment, two formulations were considered. The first formulation used was the base graph colouring problem which was generated in the same way as those in section 4.3. Problem sets were defined by the *tuple* <*n*, *m*, *c*>. We denote this first formulation to be *GC1*;

GC1: *Z*: The *n* variables

D: {1...*m*} for each variable

C: $n \times c$ randomly allocated "not-equal" constraints, where *c* is the average connectivity of the variables.

Our second formulation was generated by carrying out the same variable aggregation transformation described in section 4.5.1. This resulted in *ZDC* formulation *GC2* which is the corollary of *ZDC* formulation *R2*.

GC2: *Z*: A mixture of merged and non-merged variables. If variables v_i and v_j from *GC1* are merged, then they form a single variable $v_{i,j}$ in *GC2*, as indicated in figure 4.12.

D: Either {1...*m*} for non-aggregated variables or the set of legal tuples given by an aggregation constraint

C: A combination of original constraints and constraint which have been modified to accommodate the aggregated variables.

Sets of 100 instances were generated for two classes of graph colouring problem. For each problem, *ZDC* formulation *GC1* was solved using standard backtracking, backjumping and forward checking, all with the minimum width variable ordering. As with the previous experiment, *ZDC* formulation *GC2* was generated with the variables pre-ordered using the variable ordering before carrying out the transformation. *GC2* was then solved with the same algorithms. We also evaluated the theoretical complexity estimates for *GC1* and *GC2*.

The results of our experiments are shown in table 4.6. We use the same approach for analysing the results as we did in section 4.5.1. Table 4.6 therefore reads in the same way as table 4.5.

		Heuristic Accuracy - margin=5%			Heuristic Accuracy - margin=15%		
Algorithm + Heuristic	Problem Class	%correct	%incorrect	% no prediction	%correct	%incorrect	% no prediction
bt+mwo	<25,5,9.6>	7	93	0	7	93	0
bj+mwo	<25,5,9.6>	8	92	0	8	92	0
fc+mwo	<25,5,9.6>	39	61	0	39	61	0
bt+mwo	<50,3,4.5>	36	64	0	36	64	0
bj+mwo	<50,3,4.5>	55	43	2	53	36	11
fc+mwo	<50,3,4.5>	67	31	2	63	30	7

Table 4.6- summary of formulation comparison for graph colouring problems. 100 problem instances generated per problem class.

4.5.2.1 Conclusions

From the results in table 4.6 it is clear that there are significant errors encountered when using the complexity equations as a prediction mechanism for the relative suitability of graph colouring problems for the two formulations considered. The results for 5-colouring are particularly significant since the predictions give more than 50% as incorrect for all three algorithms, which is worse than can be expected from arbitrary selection of formulations.

We saw in section 4.4 how the accuracy of equations (4-9), (4-14) and (4-23) was least effective when used with graph colouring problems. Our conjecture there was that this is consistent with the reduction in the random nature of the constraints. The effect of this conjecture is likely to be a cause of the results seen in this experiment. This is particularly the case for the 5-colouring problems since the tightness of these constraints is lower than that of the 3-colouring problems. This means that larger regions of the constraint matrix consisting purely of goods exist in 5-colouring problems, thus exaggerating the reduction in randomness.

4.5.3 The *n*-Queens Problem

In (Nadel 1990a) different formulations of the *n*-Queens problem were used to assess the accuracy of (4-9) and (4-14) in ranking these *ZDC* formulation with respect to the actual cost of solving them. He used values of n=3, 4 and 5. In this section we carry out an experiment using three formulations similar to those used by Nadel. Our aim was to see if Nadel's observation still holds for larger values of *n*, and also to further assess the complexity estimate for backjumping given by (4-23).

The first *ZDC* formulation we used, *nQ_1*, is the representation which we used in section 4.4.2. Our second *ZDC* formulation, *nQ_2*, is based on the merging of pairs of rows on the board. This was described in detail in chapter 2 and section 4.5.1. Our third ZDC formulation is based of the transformation of *nQ_1* using the dual transformation (Dechter & Pearl 1989). This transformation was described in detail in chapter 3. The result of transforming *nQ_1* in this way is *nQ_3*;

nQ_3: *Z*: One variable, z, corresponding to each constraint in the *nQ_1*

D: For each variable, z, the domain is the set of legal compound labels defined in the associated original constraint.

C: In order to ensure equivalence between *nQ_1* and *nQ_3*, constraints are added between variables in *nQ_3* which originate from constraints in *nQ_1* having common variables stating that the identical label is used for the common variable.

For each value of *n*, we calculated the theoretical complexity for each algorithm combined with the natural lexical variable ordering, for each of the three *ZDC* formulations generated. We then solved each *ZDC* formulation, for all solutions. The results obtained are plotted in figures 4.13-4.15.

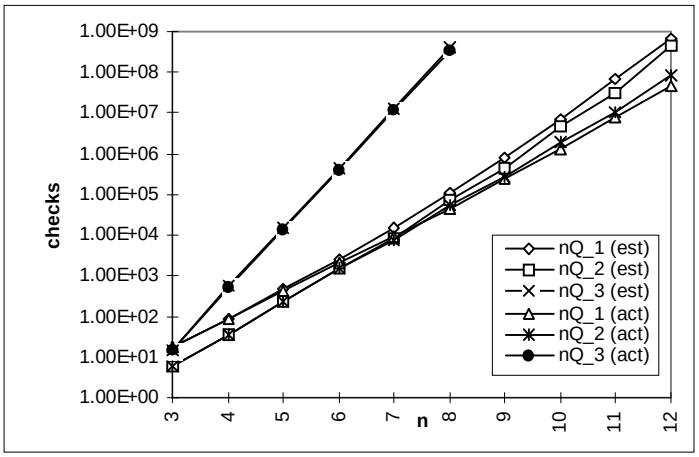

Figure 4.13 - The theoretical and actual performance of standard backtracking on the various formulations of *n*-Queens, for finding all solutions

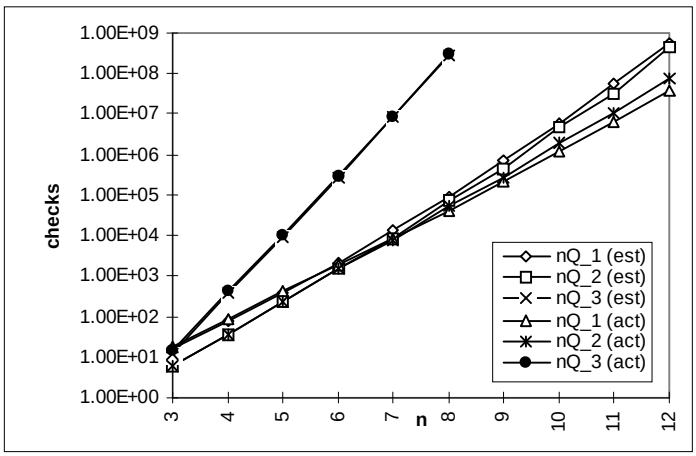

Figure 4.14 - The theoretical and actual performance of backjumping on the various formulations of *n*-Queens, for finding all solutions

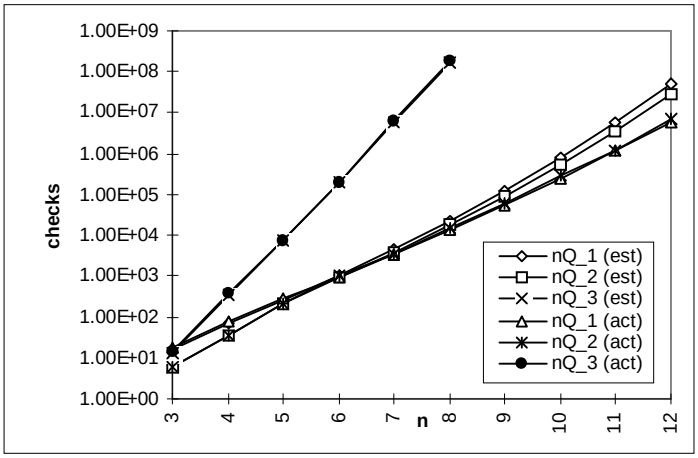

Figure 4.15 - The theoretical and actual performance of forward checking on the various formulations of *n*-Queens, for finding all solutions

As a further experiment, we also looked at how good the complexity equations were at predicting the suitability of ZDC formulations with respect to solving for a single solution only. This effectively means relaxing assumption *A4_3*. The results for forward checking are given in figure 4-16.

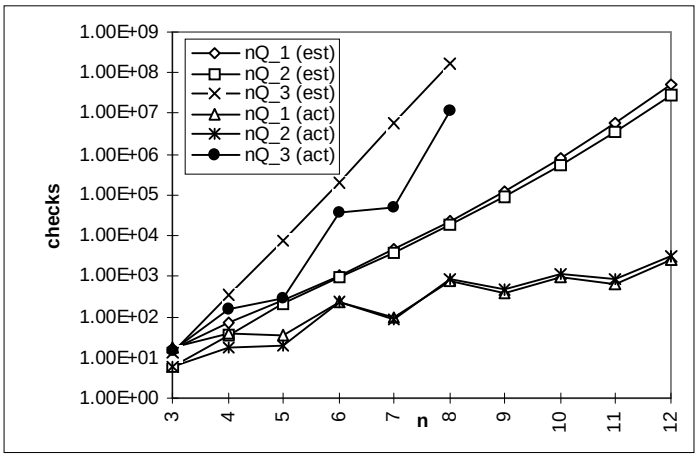

Figure 4.16 - Accuracy with forward checking when a single solution is found

4.5.3.1 Conclusions

The results in figures 4.13 - 4.16 lead to several interesting observations regarding the usefulness of the theoretical complexity estimates. The clearest of these is that formulation *nQ_3* is consistently predicted to be the costliest *ZDC* formulation to solve, for all *n* greater than 3, with a

margin over the other formulations that grows with n.[7] This is consistent with the results published in (Nadel 1990a) for standard backtracking and forward checking. Note also that the results for finding only one solution show the same ranking of the *ZDC* formulations. This suggests that relaxing assumption *A4_3* can be reasonable approach to extending the applicability of the complexity estimates.

Another observation from these results is that predictions of the relative merits of *nQ_1* and *nQ_3* are consistently incorrect. However, while this result is disappointing, the differences in the estimates for *nQ_1* and *nQ_3* are relatively small, well within an order of magnitude. Furthermore, since the actual solving costs are also very close to each other, we might expect errors in the prediction to occur, as our previous results have shown. The ramifications of an incorrect prediction for these two formulations is relatively minor as a result. In general, we can conclude that the theoretical complexity estimates produce more reliable predictions when the difference in actual complexity is large.

Finally, we see that the rankings for backjumping, based on (4-23) are comparable in quality to those for standard backtracking and forward checking. This again gives us further confidence in the correctness of (4-23).

4.6 Discussion

The primary motivation for the work in this chapter was to investigate the suitability of theory-based estimates of search complexities as a basis for *ZDC* formulation evaluation heuristics. This is attractive to us since they provide us with a relatively cheap, instance specific technique for evaluating the merits of different *ZDC* formulations of a problem. To this end we have built on previous work by first extending it to the important class of intelligent backtracking search algorithms. We then carried out a major evaluation of the effectiveness of theoretical complexity estimates.

We have investigated the accuracy of the both Nadel's and our estimates of search complexity. Accurate estimates have obvious attractions. For example, if we know before hand the expected

[7] the results for *nQ_3* for $n > 8$ are not shown since the cost of finding all solutions for bt+nat grows exponentially for values greater than 8 this becomes prohibitive. Despite this, the trend is very clear - see section 4.6

cost of searching with a range of candidate algorithms, we can choose the one most suited to the specific problem being solved. Domains of dominance are known to exists as was shown in (Tsang et al 1995). Our results show that for some classes of CSP, for example randomly generated CSPs with high density constraint graphs, these estimates can be reasonably accurate. However, we also found that for classes of CSPs having lower density constraint graphs, the accuracy of the estimates was reduced. Reducing the level of randomness in the constraint structure was also seen to affect this accuracy.

The work in this chapter also shows that theoretical estimates of search complexity can be even more effective when giving qualitative relationships of the expected search costs of different *ZDC* formulations of a problem. Our approach was shown to be useful, even for some classes of problem which have low density constraint graphs. We also noted that when the differences in actual search cost were large, the use of our estimates was particularly effective.

The use of theoretical estimates of search complexity is an algorithm specific approach. However, we have used such estimates for three different classes of algorithm. Furthermore, by demonstrating the extensibility of the method, coverage of more algorithms should be possible, thus allowing problem solvers to choose among the algorithms covered. For example there are many hybrid algorithms possible from the combination of intelligent backtracking and lookahead techniques (Prosser 1993).

We believe that our work shows how theoretical complexity estimates have a major role to play in the development of *ZDC* formulation evaluation heuristics. As we mentioned in section 4.4.1, even for classes of CSP for which the estimates are less reliable, further properties could be used in conjunction with the estimates, in order to improve the accuracy of heuristic decision making.

4.6.1 The Concept of Constancy

One further observation from our work is the concept of *constancy*. If we consider the results in section 4.5.3, we see that for *n*-Queens problems, as *n* varies, the ranking of different *ZDC* formulations changes. The important point to observe is that after a certain value of *n*, the ranking of the *ZDC* formulations becomes clear and fixed, although the differences between them may grow. In other words, after some given point we gain a *constant* ranking. If such a constant

ranking can be found, this can be extremely useful for predicting the relative merits of such problems in general. For example, we can predict with a high degree of confidence that formulation *nQ_3* is likely to continue to be a bad choice for values of *n* higher than 12.

The principle of constancy is applicable to problems which can be defined in terms of a particular parameter, provided we have enough sample estimates to reveal the trends in formulation ranking. Another example of this was seen in chapter 2 where different *ZDC* formulations of the magic series problem were solved. A constant relationship between the solving costs candidate *ZDC* formulations was observed and a clear trend was visible. We could reasonably predict in that case that larger problem sizes would fit the same pattern.

4.7 Summary

In this chapter we have made two significant contributions;

- We have extended Nadel's approach of generating theoretical complexity estimates for selected algorithms to a member of an important class of algorithms - intelligent backtracking. The results of experiments in this chapter gives us a high level of confidence in their correctness.
- We have performed a major evaluation of the usefulness of theoretical complexity estimates. Such a comprehensive evaluation has not previously been undertaken. Our evaluation has shown the potential of the approach and how it can play an important role in Z*D*C formulation selection.

In the next two chapters, we further develop the application of theoretical complexity estimates to *ZDC* formulation evaluation heuristics.