

**Efficient generic searches and
programming language
expressivity**

Robbie Daniels

s1228781

Supervised by Dr. John Longley

Master of Science
Computer Science
School of Informatics
University of Edinburgh

2016

Abstract

A search problem is of the form: *given some space S , find a point which satisfies the property P .* A generic search must locate the solution to such problems without exploiting any characteristics that they might have. It might be of surprise to the reader to learn that there are in fact several methods with which a generic search can be conducted, all of which offer considerably more efficiency than a brute-force search. An investigation of the characteristics of different generic search approaches was conducted alongside a feasibility study assessing the benefits that using a generic search framework might offer. A suite of programs capable of solving certain search problems was implemented using a generic search framework that was written by John Longley. Investigations using these implementations were then carried out in order to assess the extra computational overhead associated with the use of a generic search framework. Amongst various findings, it was found that one of the generic search approaches had an overhead that remained approximately constant when the size of the input problem was varied. The use of generic search techniques for Real number integration was then investigated. Two implementations capable of integrating a real valued function to theoretically arbitrary degree of precision were implemented, and were found to produce correct results for the example function that was tested.

Acknowledgements

I would express my gratitude to Dr. John Longley, who not only provided me with this fascinating generic search framework and many particularly interesting discussions, but also devoted his time and considerable patience to introducing me to functional programming and the more conceptually challenging aspects of this project.

Declaration

I declare that thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Robbie Daniels

Table of Contents

1	Introduction	3
1.1	Background	3
1.2	Motivation	4
1.3	Project aims	5
1.4	Organisation	5
2	Generic search methods	7
2.1	A definition of a search problem	7
2.2	The behaviour of a pruned search algorithm	8
2.3	How a pruned search might be implemented	9
2.4	Improving upon the pruned search	11
2.4.1	Continuations	11
2.4.2	The use of continuations for "resumable exceptions"	12
2.4.3	Resumable exceptions in a generic search	13
2.5	Berger's approach to the pruned approach	14
2.6	A brief summary	16
3	Some generic search problems	19
3.1	Examining the performance of a generic search algorithm	19
3.2	The N-queens problem	20
3.2.1	The overhead of a pruned search, and the advantages of the backup approach	23
3.2.2	Some unexpected behaviour	24
3.2.3	A performance comparison with an implementation in C++	25
3.3	Colouring a graph	25
3.3.1	A large caveat	33
3.3.2	Implementation-dependent performance	34
3.3.3	The behaviour of the pruned and backup pruned searches	34
3.3.4	Value instability	35
3.4	Computing 2-player Nash Equilibria	35
3.4.1	A search problem that doesn't entail back-tracking	37
3.5	The completion of partial Latin squares and Sudoku puzzles	37
3.5.1	Improving performance by adding more constraints	39
3.6	Additional comments	40
4	Generic search techniques and arbitrary precision real number integra-	

tion	41
4.1 A different type of search problem	41
4.2 A novel approach to real number integration	41
4.3 Performance and accuracy considerations	42
4.4 Implementations	42
4.4.1 A logging integrator	42
4.4.2 A limitation of the logging integrator	43
4.4.3 An integrator that makes use of continuations	43
4.5 A brief correctness check and performance comparison	44
5 Concluding remarks	47
5.1 A brief summary	47
5.2 Further work	48
Bibliography	51
A Bespoke search problem implementations	53
A.1 N-queens	53
A.2 Graph colouring	54
A.3 Graph colouring alternative implementation	55
A.4 Nash equilibrium for 2 players, bespoke implementation	56
B Real number integrator implementations	59
B.1 Logging integrator implementation	59
B.2 Continuation integrator implementation	60
C Additional code	63
C.1 Berger search implementation that was used for the investigations in chapter 3.	63
C.2 Erdős-Rényi graph generation code	64
C.3 Nash equilibrium 2D <code>isRowMax</code> and <code>isColMax</code> implementations	65
C.4 Sudoku generic search code	65
C.5 Sudoku and Latin Squares element uniqueness check code	66
C.6 Constant complexity function for cycling permutations	68
C.7 Hamiltonian circuit generic search implementation	68
D Additional data	69
D.1 Find all solutions for each search algorithm on a simple graph coloring problem	69

Chapter 1

Introduction

1.1 Background

A generic search entails searching a given search space S for points that satisfy an arbitrary property P . The methodology with which the search is conducted must also not be particular to that P . An example of a problem that can be cast as a generic search would be the completion of partial Latin squares. A Latin square is a matrix of size $n \times n$, all rows and columns of which are a permutation of the set of values $\{0, 1, \dots, (n-1)\}$. A partial Latin square is defined as above, except that it may also contain empty elements. The co-ordinates in this particular problem, given by $\sigma = (\sigma_0, \sigma_1, \dots, \sigma_{k-1})$, might be the values that are assigned to each empty cell in the partial Latin square. In this case, the corresponding property would be that every cell must be assigned a value from 0 to $(n-1)$, and that every such value must be unique to its respective row and column. In this example, each σ_i can be one of n possible values. For an example with k empty cells initially, it can then be seen that the size of the search space for this problem would be $O(n^k)$, meaning that a brute force approach would be infeasible for all but the simplest of examples. Thankfully, there exists several alternative methods of finding solutions to generic search problems, each offering a considerable performance improvement over the brute-force approach.

One such method involves the use of pruning. The execution of a generic search algorithm can be visualised using a tree, in which vertices correspond to partially complete σ , edges represent the values that the next element in σ can take and leaf nodes represent potential solutions to the problem. An example of such a tree is given by figure 1.1.

A brute force generic search algorithm fully constructs σ , or fully traverses a path from the root to a leaf. In contrast, the pruning approach considers each vertex as it encounters it, checking whether or not $P(\sigma)$ is excluded by the values $(\sigma_0, \sigma_1, \dots, \sigma_{r-1})$. Because all leaves that contain the sequence $(\sigma_0, \sigma_1, \dots, \sigma_{r-1})$ can not be solutions if this sequence is not valid, the children of the vertex corresponding to this particular sequence needn't be considered by the algorithm. In such a case, the search is reduced by $n^{(k-r)}$ leaves. It might be of surprise that a simple generic implementation of this is possible, provided that the tuple expressing σ is represented in a suitable way. An ap-

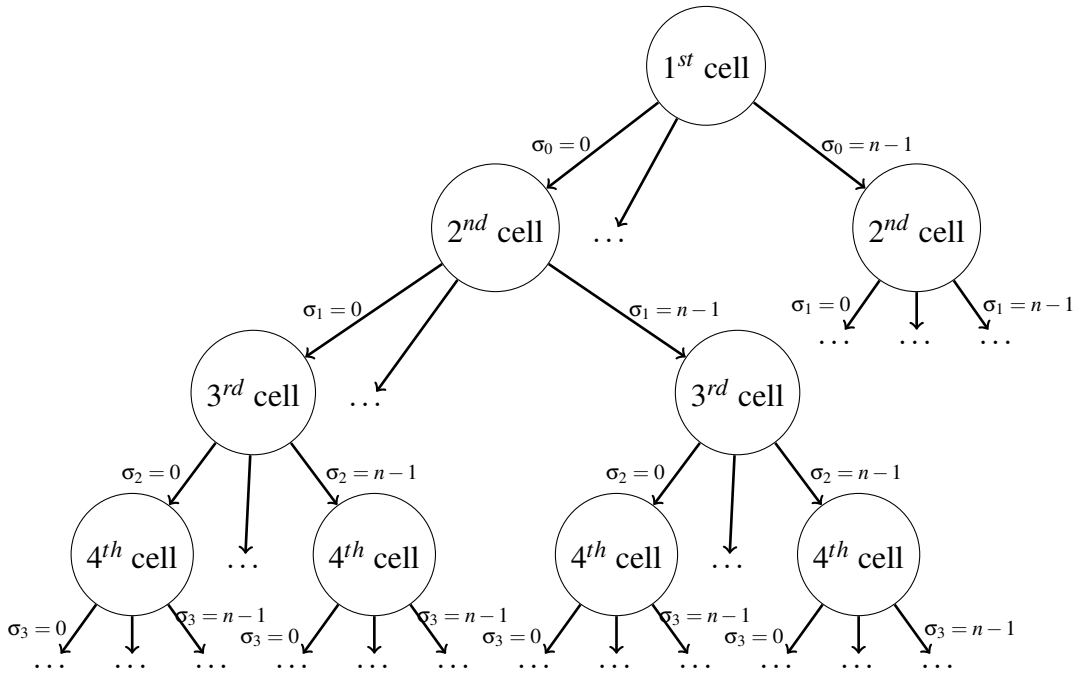


Figure 1.1: A part of the tree representing the execution of a brute force generic search.

proprately designed bespoke implementation will naturally carry out pruning anyway, however in the case of a generic search algorithm, a non-trivial concept is required.

1.2 Motivation

The completion of partial Latin squares is an example of a problem that has been shown to be NP-complete [1]. An efficiently implemented generic search algorithm may locate solutions to such problems with a comparable speed to algorithms that are allowed to exhibit none-generic behaviour. It should be noted however that in such cases, the generic search approach will not offer an improvement in performance over bespoke algorithms, it would instead be hoped that it is not much worse. The incentive to use generic search algorithms instead lies with the convenience that they can offer to a programmer. The need to locate P within some S frequently arises, in a rich variety of different settings. There exist many graph theory problems which are also in the worst case NP complete for example, such as finding Hamiltonian cycles [2] or acyclic orientations [3]. Additionally, new examples of problems that can be addressed using generic search techniques are still emerging. In [4], Escardó demonstrates that there are even some infinite sets that can be searched exhaustively. Considerable amounts of time and effort could be saved if, instead of needing to implement an efficient bespoke search (which in practice, may not even offer a significant performance improvement over a generic search algorithm anyway), an efficiently implemented generic search framework could instead be used. Problems which require several different searches to be conducted might otherwise require a considerable effort towards testing and debugging bespoke implementations.

It is important to note that the generic search methods available to a programmer are constrained by their choice of language. More powerful languages with support for ad-

vanced features such as continuations enable the implementation of the most efficient generic search algorithms.

1.3 Project aims

This report will detail a feasibility study investigating the use of advanced language features for the purpose of implementing efficient generic search algorithms, as a continuation of John Longley's work [5] [6]. A provided framework implemented in Standard ML of New Jersey by John Longley [5], which is capable of all of the approaches that will be described in chapter 2, will be used to solve a range of different problems. These implementations will be assessed against a set of corresponding bespoke implementations, which might exploit particular properties of the search problem that they seek to address. Code clarity, efficiency, flexibility and performance will constitute the assessment criteria. Performance comparisons will also be made between each generic search approach for each problem, in an effort to explore how certain characteristics of a search problem might impact the performance of each generic search method. These investigations will be run in parallel with those performed by another MSc student Kang Xia, who will be assessing the performance of some generic search implementations written in C++ and Java.

The adaptation of generic search techniques for the integration of real-valued functions will then be investigated. An implementation capable of integrating a function to an arbitrary degree of precision will be written, building upon some of the ideas that are used by the generic search techniques that will be detailed in chapter 2.

1.4 Organisation

The remainder of this thesis is structured as follows:

2. Generic search methods - The method through which problems are cast in the provided generic search framework are introduced. Different generic search algorithms are described in detail, including how they might be implemented.
3. Specific search problems - Some example problems that can be solved by a generic search algorithm are described. The performance of each search algorithm outlined in chapter 2 is then assessed on the given examples.
4. Arbitrary precision real number calculations and problems of integration
5. Concluding remarks

Chapter 2

Generic search methods

2.1 A definition of a search problem

All problems to which a generic search is applicable can be cast such that they conform to the following type signature, given in Standard ML syntax:

Listing 2.1.1:

```
1      type SearchProblem = {dimensions : int list,  
2                               property : (int->int)->bool} ;
```

The `dimensions` field constrains the shape that S may take, that is to say that if `dimensions` = $(d_0, d_1, \dots, d_{k-1})$, then S is in principle the set of tuples $(x_0, x_1, \dots, x_{k-1})$ such that $\forall i, 0 \leq x_i < d_i$. and `property` describes a function that will return true or false, depending upon whether or not $P(\sigma)$ is satisfied.

The type of the input into `property` is of particular significance. A basic data structure such as an array or list certainly represents a point in S faithfully, and for the purposes of a brute force search this is all that would be required. By instead using some function of the type $(\text{int} \rightarrow \text{int})$ however, more sophisticated behaviour is enabled on the part of the search algorithm. As an example, it may be of use to know how much of σ was inspected when $P(\sigma)$ was evaluated. Suppose σ were to be represented using an array. A wrapping function could be written, such as the one given by listing 2.1.2 (assuming that σ is in the current scope).

Listing 2.1.2:

```
1  let val inspected = array(len  $\sigma$ , false) in  
2    fn  $i \Rightarrow$   
3      (if inspected( $i$ ) = false then  
4        inspected( $i$ )  $\leftarrow$  true  
5      else  
6        () ;  
7       $\sigma(i)$ )  
8  end
```

This function behaves similarly to an `int` array, in that given an appropriate input integer, it returns a corresponding integer key. Its definition as a function however enables additional behaviour, in this particular example: using side effects to log unexplored co-ordinates. It is through akin behaviour that the generic search techniques to be detailed later in this chapter will be realised. The use of functions also leaves the choice of how σ is represented with the programmer, rather than imposing the arbitrary constraint that they must use a specific data structure.

2.2 The behaviour of a pruned search algorithm

As discussed in chapter 1, the use of pruning has the potential to vastly reduce the search space that is explored during a generic search. In order to illustrate its benefits and shortcomings more clearly, the example of completing a partial Latin Squares puzzle will be considered. The particular puzzle to be solved is given in figure 2.1.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	σ_0
2	8	7	4	1	9	6	3	5
3	σ_1	5	σ_2	8	6	σ_3	7	9

Figure 2.1: A 9x9 partial Latin square

Choosing σ to be a function enumerating the four remaining points, the corresponding tree will be similar to that given in figure 1.1, but with a height of 4. The tree will therefore have (9^4) leaves, all of which would be explored if a brute force generic search were to be used. Assuming that the row major convention was adopted for indexing the empty cells in σ , a pruning algorithm would initially discard all σ in the following set: $\{\sigma | \sigma_0 \in \{1, 2, 3\}\}$. By doing only this, the space to be searched is then reduced by 2187 leaves. In this example, $\sigma_0 = 4$ is not discarded, and so the algorithm can then start checking values for $\sigma_1 | \sigma_0 = 4$. A similar action is then repeated for the second, third and fourth empty cells, yielding a σ that "completes" the partial Latin square. If the algorithm was searching for all solutions to the partial Latin square, it would then test and discard $\{\sigma | \sigma_0 \in \{5, 6, 7, 8, 9\}\}$ (and thus, the rest of the remaining

leaves in the corresponding tree). It is of importance to note however that when a partial σ is found not to be valid, the search algorithm logs this particular point and then resumes the evaluation of the next σ from σ_0 . Because the algorithm has a record of edges that have already been traversed however, it will not traverse edges such that it will arrive at any leaf node that it has determined not to be a solution to the input problem. The tree representing the algorithm's exploration of S will therefore actually differ from figure 1.1, and is instead given by figure 2.2.

Its simplicity is deceptive however, and does not reflect the fact that it must always start the evaluation of a new σ from σ_0 . There will be overhead associated with both of these operations, and ideally they would be avoided completely (as there could be many un-necessary re-evaluations of the same points in S). It will be seen in section 2.4 how the pruned search can be adapted so that this redundancy may be avoided.

2.3 How a pruned search might be implemented

As it was suggested in section 2.1, a significant implementation detail of the pruning approach lies with how σ is represented. In the case of a naive search, the algorithm needs only to increment elements of σ until it reaches a leaf node, and the only guarantee that must be made is that the algorithm does not evaluate $P(\sigma)$ at any leaf node more than once. Such an algorithm might be realised through a recursion of the form given by listings 2.3.1 and 2.3.2.

Listing 2.3.1:

```

1  fun searchS P bounds  $\sigma$  =
2    if  $P(\sigma)$  then  $\sigma$ 
3    else searchS P bounds (increment bounds  $\sigma$  ( $\sigma$ .length))

```

Listing 2.3.2: Note that the syntax $\sigma[i]$ given in any sml pseudocode is shorthand for the sml functions: `Array.sub( $\sigma$ ,  $i$ )`, `Vector.sub( $\sigma$ ,  $i$ )` and `List.nth( $\sigma$ ,  $i$ )`, depending upon the type of σ .

```

1  fun increment bounds  $\sigma$  i =
2    if  $\sigma[i] < \text{bounds}[i]$  then
3      ( $\sigma[i] \leftarrow (\sigma[i] + 1)$  ;  $\sigma$ )
4    else if  $i \geq 0$  then
5      ( $\sigma[i] \leftarrow 0$  ; increment bounds  $\sigma$  ( $i - 1$ ))

```

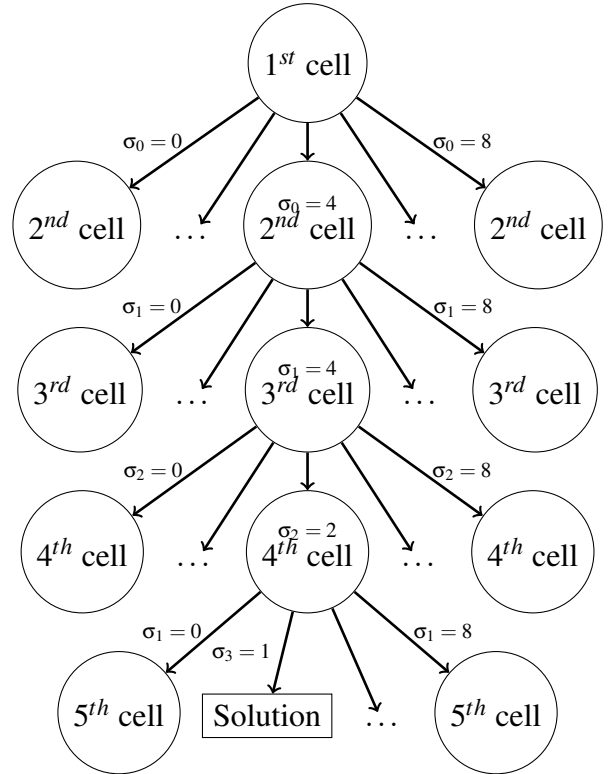


Figure 2.2: The tree representing the execution of a pruned generic search in the example given by figure 2.1, in an attempt to find all solutions. In the case where only one solution must be found, the vertices on the right hand side of the tree would not be present.

```
6      else exit
```

It could be noted here that for this recursion, σ could be any data structure capable of providing a mapping between one integer and another, but in particular might be a function of the form: $(\text{fn } i \Rightarrow \tau[i])$, where τ now denotes some array, vector or list that represent a point in S . The convention of the function σ wrapping the point τ will be adopted for the remainder of this document.

Adapting this recursion to accommodate for pruning behaviour requires several modifications. For pruning behaviour to be possible, the algorithm now needs information such that it can detect how much of σ was inspected during the evaluation of $P(\sigma)$. This is achieved by the `evalWithLog` function given in listing 2.3.5, which relies upon local state.

Listing 2.3.3:

```
1  fun searchS  $P$  bounds  $\tau$  =
2    let val (found,log) = evalWithLog  $P$   $\tau$  in
3      if found then  $\tau$ 
4      else searchS  $P$  bounds (increment bounds log)
5    end ;
```

Listing 2.3.4:

```
1  fun increment bounds [] = exit
2    | increment bounds (hd::tl) =
3      let val (i,j) = hd in
4        if  $j < \text{bounds}[i]$  then reverse  $((i,j+1)::tl)$ 
5        else increment bounds tl
6      end ;
```

Listing 2.3.5: `log` specifies the elements of σ that have been inspected, its first int equal to the index of an inspected element and the second specifying that elements value. It therefore implicitly indicates that an element has been inspected. The inspected array is not redundant however, it is included for the performance benefit that it offers. The complexity of accessing the n^{th} element in an array is $O(1)$, whereas for a list it is $O(n)$.

```
1  fun evalWithLog  $P$   $\tau$  =
2    let val inspected = array(length  $\tau$ , false)
3      val log = ref ([] : (int * int) list)
4      val  $\sigma$  = fn  $i \Rightarrow$ 
5        (if inspected[ $i$ ] then ())
6        else (inspected[ $i$ ]  $\leftarrow$  true ;
7              log :=  $((i, \tau[i])::(!log))$ ) ;
8               $\tau[i]$ )
9    in
10      ( $P(\sigma)$ , !log)
11    end ;
```

It can be seen that at a higher level, the behaviour of this pruning algorithm is largely similar to that of the naive search algorithm. Through the use of logging however, the algorithm can now determine which elements of σ did not admit $P(\sigma)$. In such cases, the algorithm marks this particular point in S to indicate that it has been searched, and then constructs and evaluates a new, un-explored σ that does not include any sequences that have already been found to be invalid.

In listing 2.3.5, σ behaves as a "thunk". This means that instead of $P(\sigma)$ being evaluated in its entirety (or "eagerly"), it is evaluated "lazily". That is to say that it is not fully evaluated until it needs to be, and is instead evaluated in parts. The thunk given by lines 4 – 8 in listing 2.3.5 also has side effects however, updating a log containing information about the current position of the search within S . Any vertices in S that do not admit $P(\sigma)$ will not be evaluated or logged, and thus the pruning behaviour that was described in section 2.2 is achieved.

2.4 Improving upon the pruned search

While a pruned search algorithm can offer a considerable improvement in performance over that of a brute force search, it was noted in section 2.2 that it still carries out some un-necessary computations. The thunk given in listing 2.3.5 will always initially evaluate $P(\sigma)$ from its first index, even if it might have already been established that this part of σ is valid. This would correspond to the algorithm traversing from the root to the leaf of the tree diagram in figure 2.2, and has the effect of unnecessarily evaluating P fully at what is almost the same σ , except that maybe its last element will differ. For example in the Latin squares example given in figure 2.1, suppose $P(\sigma)$ is evaluated, given that σ is equal to $(4, 4, 0, 0)$. σ_0 and σ_1 are both found to be valid, however this is seen not to be the case for σ_2 . This particular co-ordinate is then logged, and $\sigma = (4, 4, 1, 0)$ is then fully evaluated, in spite of the first two elements being identical to the previous value state σ . The "backup" approach remedies this limitation, by instead saving the state of the search each time that a new valid component of σ is found, and then re-commencing the construction of σ from the last valid key, should a sequence that does not admit $P(\sigma)$ be found. Referring back to figure 2.3.5 as an example, it would effectively reverse back up the tree to the lowest valid vertex from which it can start checking the new σ . This has the effect of circumventing the overhead associated with the un-necessary evaluations that the pruned approach would carry out. In order to exhibit this behaviour generically however, some non-trivial implementation details are required.

2.4.1 Continuations

It should first be noted that it would not be possible to implement the backup approach to generic searches for arbitrary P generically, without some method of storing an arbitrary state of execution. Not far removed from the infamous `goto` statement, continuations are a more "tame" feature of some modern languages that can be used to achieve precisely this end. To demonstrate the behaviour of a continuation clearly, an example will be considered. Listing 2.4.1 contains a function that recursively sums the elements of a list of integers.

Listing 2.4.1:

```

1  fun sumL [] = 0
2  | sumL (hd::tl) = hd+(sumL tl) ;

```

An equivalent function using SML of New Jersey continuations is given by listing 2.4.2.

Listing 2.4.2:

```

1  fun sumL [] k = throw k 0
2  | sumL (hd::tl) k = hd + calcc(sumL tl) ;

```

Where `throw` initialises continuation `k`, and `calcc` applies `sumL tl` to the most current incarnation of the continuation `k`. `k` is effectively behaving as an accumulator, except that any function may be applied to it. Note that for this example to work in the SML of New Jersey interpreter, `throw` should be bound to `SMLofNJ.Cont.throw`, `calcc` should be bound to `SMLofNJ.Cont.calcc`, and that `sumL` actually returns a continuation itself, and thus must also be called by `calcc` in order to obtain its result.

2.4.2 The use of continuations for "resumable exceptions"

In addition to their applications in recursive functions, continuations can also be used to encapsulate an arbitrary state of execution. It is also possible to resume computation from the point at which a continuation was thrown. Using these two ideas, listing 2.4.3 contains the declaration of a "resumable exception" by John Longley. Using this, the current state of execution can be thrown (similarly to an exception), except it is then possible to continue from this exact point at a later time.

Listing 2.4.3:

```

1  val catchcont : ('r->'s)->'t =
2    ( result : 't ,
3      arg : 'r, resume : 's->('r->'s)->'t ) Sum ;

```

Where the `sum` type is as defined in listing 2.4.4:

Listing 2.4.4:

```

1  datatype ('a,'b) Sum = inl of 'a | inr of 'b

```

The `Sum` type enables the result of a `Catchcont` call to be one of two types. The first will contain the result of a successful calculation (denoted as `'t0`), while the second will account for any case in which another resumable exception is needed. The function `('r->'s)` is in effect a "dummy function", encapsulating the passing of a value into some other level of execution, some arbitrary sequence of operations being carried out on said value, and the result from this being returned. This effectively allows for any function of the type `('r->'s)->'t` to be "thrown" as a continuation to another level of execution, similarly to an exception. This continuation can then later be evaluated by `catchcont`, which will return either the result of the computation that was thrown, or in the case where more information is required to return a result, another resumption along with the `'r` that was used in that evaluation.

The implementation of `catchcont` is complex, using both continuations and state to achieve the behaviour that has been described. Because of this, the details of its implementation will not be included.

2.4.3 Resumable exceptions in a generic search

As with the case of the pruned search, the recursion given in listings 2.3.1 and 2.3.2 can be modified in order to exploit resumable exceptions. This new recursion will differ however, in that instead of maintaining an array or log of values to represent the σ to be examined, it will instead use a stack, of the type given in listing 2.4.5 and a `resumption` is defined as follows: $(\text{int} \rightarrow \text{int}) \rightarrow (\text{bool})$. For the code in listing 2.4.6, the initial value of the `resumption` should be: $\text{fn } \sigma \Rightarrow P(\sigma)$.

Listing 2.4.5: In this stack, each entry contains an integer to specify an element i in σ , another integer to specify the value of that element (j), and a function that given some value k , will return a new `resumption` in which the i th element of σ is equal to k .

```
1    val restartStack = ref ([] : (int * int * (int->resumption)) list)
```

Listing 2.4.6:

```
1    fun searchS bounds (res : resumption) v =
2      let val (found, v') = evalWithCont res v in
3        if found then v' else
4          (increment bounds v';
5           let val (_, j, res) = currentResumption() in
6             SearchS bounds (res j) v'
7           end)
8      end ;
```

Note that similarly to τ , v denotes a point in S . For each co-ordinate that it contains however, a boolean value indicates whether or not that co-ordinate has been inspected by the search algorithm.

Listing 2.4.7:

```
1    fun increment bounds v =
2      let val (i, j, resumption) = popRestartStack() in
3        if j < bounds[i] then
4          (pushRestartStack (i, j+1, resumption) ;
5           v[i]  $\leftarrow$  (j+1, true));
6        else
7          (v[i]  $\leftarrow$  (0, false) ;
8           increment bounds v
9        end ;
```

The `popRestartStack` and `pushRestartStack` functions enact the standard pop and push stack operations upon `restartStack`, while `currentResumption` returns the head of this stack without removing any of its entries.

There is an important distinction between how `increment` behaves in listing 2.4.7 and

in listings 2.3.2 and 2.3.4. While still being responsible for enforcing the boundaries imposed by the dimensions of S , it is now also responsible for storing a corresponding resumption. Due to the extra information it is now responsible for handling and how this information must be accessed, its results can exclusively be side effects.

The `evalWithCont` function, given by listing 2.4.8 is a critical component in this implementation of the backup pruned approach, being responsible for integrating the use of resumable exceptions into the evaluation of $P(\sigma)$.

Listing 2.4.8:

```

1  fun evalWithCont resumption v =
2    case catchcont(fn f =>
3      resumption (fn i =>
4        case v[i] of
5          (j,true) => j
6          | (_,false) => f(i) ))
7    of
8      inl answer => (#result answer, v)
9    | inr cont => let val i = #arg cont in
10      (v[i] <- (0,true) ;
11        PushRestartStack (i,0,#resume cont) ;
12        evalWithCont (#resume cont 0) v)
13    end ;

```

The anonymous function in lines 2 – 6 of listing 2.4.8 will return an already determined value for j when given values of i that allow $P(\sigma)$, otherwise it will test to see if the input value for i admits $P(\sigma)$. The two outcomes that can arise from an evaluation of `catchCont` are then accounted for by lines 9 – 12 of listing 2.4.8. Firstly, if all elements of σ have been considered in an evaluation of $P(\sigma)$, a result can be returned. Otherwise a continuation is returned, with which the algorithm may resume from this particular stage in the evaluation of $P(\sigma)$. It is important that from this resumption, the algorithm is capable of exploring all possible values for j , given the current value of i . As a result, the resumption is stored with a value of j equal to 0. The recursive call to `evalWithCont` will evaluate an extra digit every time it is called. Line 8 ensures that it will be called either until a boolean result (true or false) is finally attained. This means that `evalWithCont` will fill σ with as many digits as is required to determine whether or not $P(\sigma)$ will evaluate to true.

In reference to figure 2.2, the behaviour described above would correspond to the algorithm considering a particular none-leaf vertex and then saving the current position in the tree. This position is then the start point of the next evaluation of $P(\sigma)$, in contrast to the pruned approach, which would instead begin from the root in this particular case.

2.5 Berger's approach to the pruned approach

It may be to the reader's surprise that there is in fact a generic search algorithm which exhibits similar behaviour to that of a pruning algorithm, however it is completely functional (and thus requires no state). In contrast to the other algorithms presented in

this chapter so far, it is only directly capable of finding one solution to a given search problem, rather than all of them. This approach uses ideas behind the "fan functional", which returns the maximum depth to which S must be explored in order to evaluate P (this value is known as the "modulus of uniform continuity"). Its recursion takes the form given in listing 2.5.2, and is known as the "Berger search" [7].

Listing 2.5.1: This is the implementation of the `memoize` function called in listing 2.5.2. It should be noted that this does indeed make use of state, however it is required in order to simulate the "call by need" behaviour that is required by the Berger search algorithm. This is because all expressions in `sml` are evaluated "eagerly", that is to say that expressions are evaluated immediately rather than only when their results are needed. Call by need or "lazy" evaluation only evaluates expressions when their results are actually required, and is a feature of other functional programming languages such as Haskell. This function is therefore considered to behave functionally, for the purposes of these investigations.

```

1  fun memoize (thunk : unit → 'a) =
2      let val cache = ref (NONE : 'a option) in
3          fn () ⇒ case !cache of
4              SOME x ⇒ x
5              | NONE ⇒ let val x = thunk() in
6                          (cache := SOME x) ; x
7                      end
8      end ;

```

Listing 2.5.2: Note that as with in `sml` syntax, `#[]` represents an empty vector.

```

1  fun searchS p bounds =
2      let fun bestPartialFrom fullSolution [] _ = fullSolution
3          | bestPartialFrom partSolution (currentBound::restOfBounds) j =
4              let val partSolution' = concat partSolution #[j] in
5                  if j = currentBound-1 then
6                      bestPartialFrom partSolution' restOfBounds 0
7                  else
8                      let val σ = bestFunctionFrom partSolution' restOfBounds in
9                          if p(σ) then σ
10                         else
11                             bestSolutionFrom partSolution (currentBound::restOfBounds)
12                                                         (j+1)
13                         end
14                  end
15      and bestFunctionFrom partSolution bounds =
16          let val τ = memoize (fn () ⇒
17                                  bestPartialFrom partSolution bounds 0)
18          in
19              fn i ⇒ if i < length partSolution then partSolution[i]
20                      else τ()[i]
21          end

```

```
22   in
23     bestPartialFrom #[] bounds 0
24   end ;
```

The recursion given in listing 2.5.2 behaves as follows. Given an input list to represent σ (which will initially be empty), it appends an element with value j at line 4. This would be equivalent to traversing an edge to a new vertex in any of the tree diagrams previously given. Line 5 then enforces the boundaries imposed by the shape of S upon the value that can be taken by j , such that if it is at its maximum value, the algorithm will start considering the next element of σ , or again traverse an edge to a new vertex in the corresponding tree representing the search. If j has not reached its maximum value however, a function representing σ is recursively constructed with a call to `bestFunctionFrom` at line 8. Given a partial solution to the input problem, `bestFunctionFrom` will effectively attempt to pad this solution with a set of values which might also admit $P(\sigma)$. This set of values will fall under one of two categories. The first is that they will be such that $P(\sigma)$ evaluates to true, in which case a solution has been found. The second case is that the last element of σ has not taken its maximum value, but $P(\sigma)$ has evaluated to false. In this case, line 11 will iterate the originally input index of `partSolution` that was input into the current call of `bestPartialFrom`, and so a new branch of S will be explored (and the previously inspected one will be "pruned" away). The pattern matching in lines 2, 3, 6 and 11 implicitly control the index of σ that is currently being incremented and tested, and so in the case where `partSolution` is equal to the length of σ , line 2 ensures that the algorithm will not continue searching beyond the dimensions of S (and thus, the algorithm will terminate).

The `bestFunctionFrom` function will effectively select a σ , the length of which will be equal to the modulus of uniform continuity. This σ will not necessarily be a solution, however. Because of this, the evaluation of $P(\sigma)$ at line 9 is required. It is interesting to note that as a consequence of the behaviour of `bestFunctionFrom`, even in the case where S is an infinite tree, provided that enough of a point in S has been specified, further recursive calls to `bestPartialFrom` will not be triggered.

2.6 A brief summary

The naive approach simply enumerates and explores every leaf of S , without making any attempt to avoid un-necessary evaluations of $P(\sigma)$. As a result of this, it has a complexity of n^k for a search problem in which σ has k elements which can take n possible values. Through the use of local state, the pruned approach effectively constructs σ one element at a time, determining each time whether the value of the current element admits $P(\sigma)$. By preemptively establishing that a particular sequence of values will make $P(\sigma)$ evaluate to false and discarding this sequence, the search is reduced by $n^{(k-r)}$ leaves, where $r - 1$ denotes the index of the element in σ that was found to be invalid. This method still carries out un-necessary computations however, as it will always evaluate $P(\sigma)$ from the beginning, in spite of the fact that in the majority of cases, only the last element of σ has been changed. The backup approach rectifies this limitation with the use of continuations, which enable the evaluation of $P(\sigma)$ from a more appropriate element. Finally, the Berger approach is a completely functional

searching technique that exhibits behaviour that is similar to a pruned search. In practice, the avoidance of state comes at the cost of lower performance than that of an algorithm implementing the pruning approach however.

Chapter 3

Some generic search problems

3.1 Examining the performance of a generic search algorithm

A particularly rich assortment of problems can be cast as a generic search, many of which having characteristics that will influence the performance of the generic search algorithm attempting to solve them. This chapter will examine several such problems according to their nature, and how this might affect the efficiency with which they are solved when using the methods described in chapter 2.

In order to investigate how the characteristics of different search problems might affect the performance of a generic search algorithm, several different search problems were considered. They will be described in more detail later in this chapter.

The inclusion of the Berger search algorithm in the investigations necessitated the performance comparison to be based upon the amount of time required by each search algorithm to find one solution to their input problem. Moreover, there were some problems that were investigated (such as N-queens, which will be described in section 3.2) for which there could be many solutions. Basing the performance comparison upon the amount of time to find all solutions in this case could have been impractical even for smaller examples, and so it would not have been possible to investigate a wide range of different input problem sizes. A wide range of examples for each problem were investigated, so that it could be seen how the performance of each generic search algorithm scaled with the size of the input problem.

For some of the problems that were investigated, a corresponding "bespoke" algorithm was implemented. Contrarily to what the name might suggest, these bespoke algorithms were not necessarily designed to exploit any properties that might be particular to a search problem. They were instead designed merely to avoid the additional computational expense that is inherent in a generic search algorithm, so that this overhead could be investigated and quantified. It was therefore important that there was a degree of uniformity in the behaviour across the bespoke algorithm implementations. While this was certainly aimed for however, it should be noted that this was not possible for all search problems that were considered. The generic search counterparts to these

bespoke algorithms were all implemented on top of the generic search framework that was provided by John Longley [5] (which implemented all of the search methods that were detailed in chapter 2), and were written to obey the type signature given by listing 2.1.1.

The overhead associated with the use of each generic search algorithm was quantified by calculating their timing values as a multiple of those of a corresponding bespoke implementation for a given problem. The use of these ratios also enabled for a direct comparison to be made between different sized instances of a given problem in the case where one solution was being found. Such a comparison would otherwise not be meaningful, as for many search problems, the amount of time taken by a search algorithm to find one solution will not be linear with n .

In order to get accurate timing data that also reflected the amount of time taken by the `sml` garbage collector, these investigations were conducted using the `SMLTimer` struct.

Preliminary investigation demonstrated that the availability of system resources could impact upon the timing results, resulting in noise that could have an amplitude as large as 40 microseconds. This was found to arise even when the computer that was used was otherwise idle. In order to remedy this, smaller examples were averaged over repeat runs upon identical inputs. The amount of repeats used in the averaging required to reduce this noise depended both upon the search problem that was considered, and its size. The amount of runs used for each average was generally increased until this noise phenomenon was no longer apparent. Using averaging in this way enabled for smaller examples to be considered during the investigations.

Additionally it was found that in all examples that were considered that the timing information was dominated by `CPU var` time spent outside of the garbage collector, and the time spent in the garbage collector was found to be completely negligible. Because `CPU sys` time is more related to I/O and system calls (and thus does not reflect the performance of the parts of the algorithm that are responsible for conducting the generic search), only `CPU var` time was considered, and no comparisons were made based upon the amount of time that the interpreter spent in the garbage collector. Finally, it was also found that in the vast majority of cases that the naive search was several orders of magnitude slower than the other generic search implementations, and thus was not considered in these experiments unless stated otherwise.

The rest of this chapter will introduce the search problems that were considered. Some data that was generated to investigate the performance of each generic search algorithm on each of these search problems will then be presented and discussed. Please note that the convention of s representing σ will be used throughout the listings given in this chapter. Additionally, only valid `sml` syntax will be used.

3.2 The N-queens problem

The N-queens problem entails finding a configuration of queen pieces on an $N \times N$ chess board, such that no queen can be captured in one move under the rules of chess. Cast as a generic search problem, the k^{th} element of an input σ might index the column of the k^{th} queen, while the corresponding key might give that queen's row. The

property in this case would then be that σ must be a permutation, and that no two elements of σ must allow for any two queens to be captured diagonally. An example implementation of this that was provided by John Longley [5] is given in listing 3.2.1.

Listing 3.2.1:

```

1  fun testQueenPair s i j =
2      let val (jpos,ipos) = (s j,s i) in
3          not (ipos = jpos orelse ipos-jpos = i-j
4              orelse ipos-jpos = j-i)
5      end ;
6
7  fun testQueenPairsFrom s i j =
8      j=i orelse (testQueenPair s i j
9          andalso testQueenPairsFrom s i (j+1)) ;
10
11 fun testQueensFrom n s i =
12     i=n orelse (testQueenPairsFrom s i 0
13         andalso testQueensFrom n s (i+1)) ;
14
15 fun nQueens n : SearchProblem =
16     { dimensions = List.tabulate (n, fn _  $\Rightarrow$  n) ,
17       property = fn s  $\Rightarrow$  testQueensFrom n s 0 } ;

```

The property field of this SearchProblem implementation uses recursion to make the following guarantees about each element of s , in turn. No two elements of σ must be equal, as σ would no longer be a permutation (and two queens would occupy the same row). The difference between the row indexing of each queen must also be different from the difference between the column indexing of each queen, as this would otherwise correspond to two queens occupying the same diagonal.

N-queens was one of the problems that was considered in the investigation into the performance overhead associated with the use of a generic search framework. Values calculated for each $n < 17$ were averaged over as many as 100,000 repeats for faster implementations, as preliminary investigation demonstrated that in the case of the faster implementations, the noise phenomenon described in section 3.1 had a significant effect upon timing results. In cases where $n \geq 17$ values were instead averaged over fewer runs, as the noise issues were found to be considerably less significant.

It should also be noted that for every value of n , it was checked that each algorithm had located the same solution. This was done to ensure that they had moved through S similarly, as otherwise their timings and ratios would not be comparable for the purposes of investigating the performance overhead of each implementation.

Figure 3.1 illustrates the performance of the different generic search implementations as a multiple of the corresponding bespoke implementation written by John Longley, given in appendix A.1. These results are also summarised in table 3.1.

n	Berger	Pruned	Backup pruned	Bespoke implementation absolute time (μs)
9	112	50	30	8
10	91	42	13	38
11	98	33	17	22
12	91	38	12	157
13	105	42	12	62
14	107	64	14	1141
15	103	60	12	1088
16	142	76	14	7627
17	221	94	18	3356
18	245	85	18	28763
19	287	84	19	1888
20	285	81	12	183117
21	218	51	7	12309
22	334	90	12	1899755
23	437	100	19	25568
24	407	107	12	517464
25	429	88	9	68796
26	494	138	11	561975
27	543	139	11	684966
28	595	127	11	4900192
29	644	122	10	2626448
30	641	95	10	110546417

Table 3.1: The time spent by each generic search algorithm to find one solution to the N-queens problem, as a multiple of that of the bespoke N-queens search algorithm that was used, for different problem sizes (n). All ratios have been rounded to the nearest integer, for clarity.

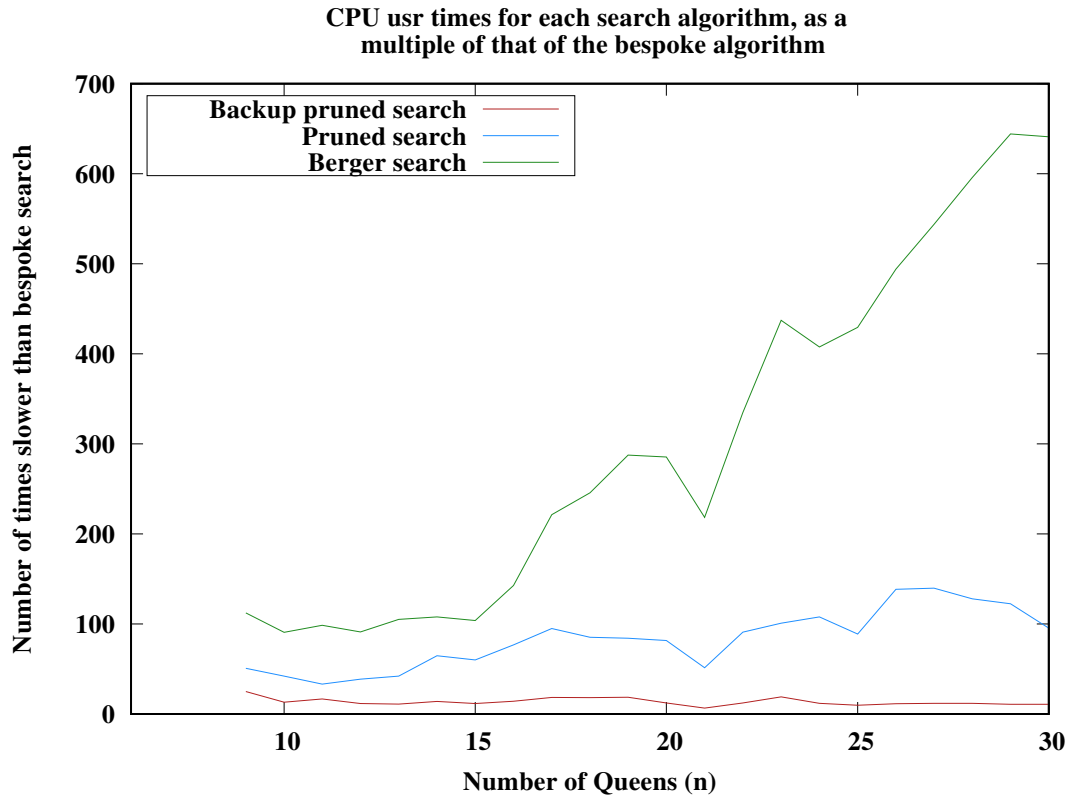


Figure 3.1: The time spent by each generic search algorithm to find one solution to the N-queens problem, as a multiple of that of the bespoke N-queens search algorithm that was used, for different problem sizes (n). Note the almost constant scaling of the pruned and backup pruned searches, as well as the dips that appear to occur across all generic search methods for certain n .

3.2.1 The overhead of a pruned search, and the advantages of the backup approach

Figure 3.1 demonstrates that the generic search algorithm exploiting resumable exceptions offered a considerable improvement in performance over algorithms based upon the other generic search approaches. Even in the case of small problems, the backup pruned search offered a two to three times speed up over the pruning algorithm. The benefits of resumable exceptions become particularly apparent for larger values of n however, as the timing ratios of the backup pruned search implementation remain in the range 7 to 19.

This is in contrast with the pruned search, the ratios of which appear to grow approximately linearly with the size of the problem. This might be due to the limitation of the pruned search that was discussed in chapter 2, in which for all σ , an algorithm implementing the pruned approach will evaluate $P(\sigma)$ from the first element of σ . The size of the problem was increased linearly in this investigation, and so the linearly increasing size of σ will result in a linear growth in the amount of time required by a pruned search implementation to evaluate $P(\sigma)$. In the case of the backup pruned approach

however, $P(\sigma)$ is not always fully evaluated. It would therefore be expected that the linear ratio growth seen with the pruned search would not be present for the backup pruned search algorithm.

3.2.2 Some unexpected behaviour

It was stated at the beginning of this chapter that the amount of time required by a search algorithm to locate one solution to a search problem will not necessarily be linear with the size of that problem, and indeed N-queens is one such problem. Table 3.1 presents many examples of n such that the absolute time required for the bespoke search implementation to find one solution has decreased when n has been incremented. In order to remedy this and enable a direct comparison of the values that were obtained therefore, all timings were calculated as a multiple of those of the bespoke N-queens algorithm. In spite of this, figure 3.1 indicates several dips in timings that were common across all search methods (for example when $n = 21$).

Table 3.1 indicates that the particularly significant dips at $n = 21$ and $n = 25$ occurred when the absolute time taken by the bespoke algorithm to find one solution had also dropped significantly. The decrease in the ratios of each generic search algorithm at these points could be attributed to an inappropriate number of repeat computations for the average timings. An alternative explanation might be that in cases in which the bespoke algorithm found a solution a shorter amount of time, the average depth to which each search algorithm had to traverse S before finding a solution might have also been lower than usual. In such a case, one might expect to see less of a performance overhead associated with the more computationally expensive backtracking behaviour of the generic search algorithms, and so the generic search approaches will perform more favourably in comparison to a bespoke implementation.

A combinatorial analysis by John Longley suggested that the pruned search implementation should be approximately twice as fast as the Berger search, for any given problem. Figure 3.1 appears to suggest the contrary however, even in the cases where n was small and the ratios of each search algorithm were comparatively closer to each other, and trended in a similar fashion. Figure 3.1 suggests that the growth of the Berger search ratios might be quadratic. This extra growth might in fact be explained by the way in which the Berger search was implemented. The full implementation of the Berger search that was used is contained in appendix C.1. It differs from listing 2.5.2, which was intended to serve as pseudocode.

Note that on line 25 of appendix C.1, there is a call to the `Vector.concat` function. The Standard ML Basis Library documentation on the `vector` structure states that the `vector` type is immutable, and thus can only be copied. This is reflected by their documentation on the `Vector.update` method, which will create a new vector, rather than directly mutating its own input[8]. With this in mind, while it is not explicitly stated in the documentation, it is plausible that the `Vector.concat` function will do an element-wise duplication of each `vector` in its input `vector list`. This will have a best case complexity of $O(n)$ (where n denotes the combined length of all input vectors), as it must address every element of each vector. This operation will be carried out almost every time the function `bestShot` is called, and so it is also plausible that

this additional $O(n)$ operations may result in the overall complexity of this particular Berger search implementation being $O(n^2)$, rather than the expected $O(n)$. This may certainly explain why the Berger search ratios appear to scale quadratically with the size of the input search problem.

A possible remedy to the inefficiency described above would be to use a mutable `sml` array in conjunction with array slices in the place of vectors, however this use of state would then mean that the Berger search algorithm would no longer be purely functional.

3.2.3 A performance comparison with an implementation in C++

In [9], the generic pruned search algorithm was implemented in C++, and its performance was evaluated using the N-queens problem. Table 5 in that document demonstrates similar performance between the bespoke `sml` implementation used in this investigation, and his own bespoke C++ implementation. The `sml` pruned and backup pruned implementations appeared to offer considerably better performance than the C++ pruned implementation however, which had considerably larger ratios than those seen in table 3.1. The ratios of the Berger search would appear to be more comparable, having a similar rate of growth for all values for n that can be compared. This might suggest that there are un-necessary computations causing the timings of the C++ implementation to scale similarly to those of the Berger search Implementation. Alternatively, it might just be the case that the language features that were required to implement a pruned search generically in C++ resulted in there being more overhead than there was in the case of the corresponding `sml` implementations.

3.3 Colouring a graph

Graph colouring is the problem of finding a way of colouring some input graph g such that no two vertices that are connected by one edge are the same colour. For these investigations, the problem was cast such that the indices of σ enumerated the vertices of g . The value at each index of σ indicated the colour of that particular vertex. For this example, the property was therefore that if there was found to be an edge connecting two vertices, those two vertices must not have the same value assigned to them in σ . There is a logical equivalent of this however which was also implemented as a property: if two vertices are the same colour, there must not be an edge connecting these two vertices. The two versions of this `SearchProblem` that were implemented are given in listings 3.3.3 and 3.3.4. **Listing 3.3.1:**

```
1 type Graph = { size : int,  
2               isEdge : int→int→bool } ;
```

It is important to note that the `Graph` type described in listing 3.3.1 has the scope to impact the performance of any search algorithm that makes use of it, due to the `isEdge` function. In the case of the experiments to follow, all `isEdge` implementations were all instances of the function given in listing 3.3.2.

Listing 3.3.2:

```
1 fun isEdgeAdjList adjList =
```

```
2   fn i  $\Rightarrow$  fn j  $\Rightarrow$  List.exists (fn k  $\Rightarrow$  k=j) (List.nth (adjList, i)) ;
```

It can be seen that the function given by listing 3.3.2 assumes that the input graph is an adjacency list, and will therefore have a complexity of $O(n)$.

Listing 3.3.3:

```
1  fun isColoring g k : SearchProblem =
2    let fun testColoring (g : Graph) (p : int $\rightarrow$ int) i j =
3      if (#isEdge g i j) then not ((p i) = (p j)) else true;
4
5      fun testVertexPairsFrom g p i j =
6        i=j orelse (testColoring g p i j
7          andalso testVertexPairsFrom g p i (j+1)) ;
8
9      fun testVerticesFrom g p i =
10       i=(#size g) orelse (testVertexPairsFrom g p i 0
11         andalso testVerticesFrom g p (i+1)) ;
12
13  in
14    { dimensions = List.tabulate ((#size g), fn _ => k) ,
15      property = fn p => testVerticesFrom g p 0 }
16  end ;
```

In listing 3.3.3, the property function checks each element of σ to see if it is connected to any other element of σ by an edge in g , and if so, that the two elements have different values. This directly corresponds to checking each whether any neighbouring vertices in g are the same colour.

Listing 3.3.4:

```
1  fun isColoringAlt g k : SearchProblem =
2    let fun testColoring (g : Graph) (p : int $\rightarrow$ int) i j =
3      not (if ((p i) = (p j)) then (#isEdge g i j) else false);
4
5      fun testVertexPairsFrom g p i j =
6        i=j orelse (testColoring g p i j
7          andalso testVertexPairsFrom g p i (j+1)) ;
8
9      fun testVerticesFrom g p i =
10       i=(#size g) orelse (testVertexPairsFrom g p i 0
11         andalso testVerticesFrom g p (i+1)) ;
12
13  in
14    { dimensions = List.tabulate ((#size g), fn _ => k) ,
15      property = fn p => testVerticesFrom g p 0 }
16  end ;
```

The code given in listing 3.3.4 is very similar to that in listing 3.3.3, however it instead asserts that if two vertices are the same colour, that they are not connected by an edge.

The performance differences of these two implementations will be demonstrated and discussed later in this section.

Preliminary investigations verifying the correctness of these two implementations were carried out. They included the colouring of a basic four vertex graph (containing only one cycle, so that a quadrilateral was formed). It was then verified that each implementation could find a colouring of a planar map of Europe that obeyed the four colour theorem, where a planar graph g is such that the constituent vertices of g can be positioned such that no edges overlap each other, and the four colour theorem states that a planar graph can be coloured with only four colours [10]. All preliminary investigations produced their expected results, with a caveat that will be discussed in section 3.3.1.

The investigations that were conducted entailed the timing of each graph colouring method with each generic search method and then calculating them as a multiple of their corresponding bespoke implementation timings. The code that was written for each bespoke implementation is included in appendices A.2 and A.3.

The graphs used for these investigations were generated using the Erdős-Rényi model, which parametrises a graph g by its number of vertices n and an edge addition probability $p \in [0, 1]$ [11]. The code that was written to generate graphs according to the Erdős-Rényi model is given in appendix C.2. All timings were averaged over an increasing amount of identical runs, until they stopped demonstrating instability due to the noise phenomenon described in section 3.1. Inputs were uniform both across the different search methods that were used and across the different graph colouring `property` implementations. Finally, three different values for p were considered. $p = 0.2$ was investigated as an example of a graph for which a colouring should be found quite quickly, due to the fact that it will not be computationally expensive to find a colouring for a sparse graph. Graphs generated with $p = 0.5$ were also included, as such graphs should be more computationally expensive to find solutions for, and it was expected that it would not be possible to colour all of them with four colours. The third value for p to be considered was 0.8. Graphs generated with this value will be dense, and thus it would be quite likely that it would not be possible to colour them with four colours. In every experiment that was carried out, the search algorithms were restricted to using four colours so that this chance of a graph having a solution might be controlled.

Figures 3.2, 3.3 and 3.4 present the timings of both of the search problem implementations that were described earlier in this section, for different sized graphs according to the Erdős-Rényi model, as a multiple of that of their corresponding bespoke search implementations. Tables 3.3, 3.4, 3.5, 3.6, 3.7 and 3.8 contain the values shown on each of these graphs.. Table 3.2 indicates which graph of the graphs that were used had colourings.

n	$p = 0.2$	$p = 0.5$	$P = 0.8$
4	Y	Y	Y
5	Y	Y	Y
6	Y	Y	N
7	Y	Y	N
8	Y	Y	N
9	Y	Y	N
10	Y	Y	N
11	Y	N	N
12	Y	N	N
13	Y	N	N
14	Y	N	N
15	Y	N	N
16	Y	N	N
17	Y	N	N

Table 3.2: This table indicates which of the graphs that were considered in these experiments had solutions.

n	Berger	Pruned	Backup pruned	Bespoke implementation absolute time (ns)
4	2.6	1.3	1.1	1232
5	3.3	1.5	1.0	1300
6	4.7	1.4	1.1	2119
7	3.7	0.8	0.7	2994
8	4.7	2.0	1.1	3160
9	6.1	1.7	1.0	4457
10	6.0	2.1	1.0	6320
11	6.5	1.7	1.0	6046
12	9.1	3.1	1.1	7266
13	10.6	3.4	1.1	8563
14	13.5	4.0	1.2	8394
15	14.0	2.2	1.3	10240
16	14.4	6.9	1.7	14990
17	14.7	6.3	1.3	21309

Table 3.3: The amount of time taken by each search using the first graph colouring implementation, as a multiple of that of the corresponding bespoke search. The example graphs were generated using the Erdős-Rényi model for different values of n , where $p = 0.2$.

n	Berger	Pruned	Backup pruned	Bespoke implementation absolute time (ns)
4	2.2	1.1	1.4	659
5	2.8	1.1	1.5	866
6	3.4	1.7	1.8	1273
7	3.9	1.5	1.6	1319
8	5.8	2.0	1.8	1380
9	7.0	2.5	1.9	1953
10	7.0	3.3	2.0	2446
11	8.6	2.8	1.9	2306
12	10.4	4.3	2.1	2759
13	10.5	3.8	2.1	3513
14	13.8	4.7	2.1	3599
15	14.4	3.8	1.8	5506
16	15.6	3.3	1.9	5553
17	15.6	6.8	1.9	6673

Table 3.4: The amount of time taken by each search using the second graph colouring implementation, as a multiple of that of the corresponding bespoke search. The example graphs were generated using the Erdős-Rényi model for different values of n , where $p = 0.2$.

n	Berger	Pruned	Backup pruned	Bespoke implementation absolute time (ns)
4	2.6	2.1	5.0	1924
5	3.0	1.5	2.8	2620
6	5.3	2.8	2.2	2701
7	6.8	4.8	2.2	3080
8	8.3	3.6	1.5	3880
9	9.7	4.1	1.5	11809
10	9.8	3.3	1.5	5540
11	9.0	5.2	1.3	19664
12	14.1	6.5	1.2	3125492
13	20.9	5.4	1.4	924510
14	28.5	4.5	1.6	349112
15	24.8	5.9	1.0	4812223
16	26.9	6.3	1.1	2814561
17	34.9	5.7	1.2	7399060

Table 3.5: The amount of time taken by each search using the first graph colouring implementation, as a multiple of that of the corresponding bespoke search. The example graphs were generated using the Erdős-Rényi model for different values of n , where $p = 0.5$.

n	Berger	Pruned	Backup pruned	Bespoke implementation absolute time (ns)
4	1.6	1.3	1.4	2059
5	3.3	2.0	1.8	1473
6	4.1	3.0	2.3	1633
7	4.9	3.9	2.7	1812
8	6.1	3.5	2.4	2095
9	8.7	4.9	2.7	4634
10	7.2	3.0	2.1	2692
11	9.5	6.4	2.6	6969
12	16.4	8.0	2.7	1092102
13	21.0	6.2	2.8	355867
14	30.9	5.5	3.3	129193
15	27.2	7.3	2.4	1634371
16	31.5	7.8	2.3	946269
17	40.3	7.2	2.4	2426894

Table 3.6: The amount of time taken by each search using the second graph colouring implementation, as a multiple of that of the corresponding bespoke search. The example graphs were generated using the Erdős-Rényi model for different values of n , where $p = 0.5$.

n	Berger	Pruned	Backup pruned	Bespoke implementation absolute time (ns)
4	2.8	1.6	1.7	1861
5	4.0	2.3	1.7	2169
6	5.0	3.4	2.0	44201
7	6.1	3.7	1.8	66788
8	9.7	3.2	1.9	48481
9	11.8	3.9	1.7	149510
10	14.7	3.4	1.8	92412
11	18.6	3.7	1.9	79105
12	21.7	3.6	1.9	68743
13	21.1	3.0	2.0	29857
14	26.1	3.6	1.8	73584
15	32.0	5.0	1.6	156268
16	32.8	3.6	1.8	75076
17	33.6	3.6	1.7	99223

Table 3.7: The amount of time taken by each search using the first graph colouring implementation, as a multiple of that of the corresponding bespoke search. The example graphs were generated using the Erdős-Rényi model for different values of n , where $p = 0.8$.

Search algorithm performance ratios from finding colourings for Erdős-Rényi graphs, generated with $p = 0.2$

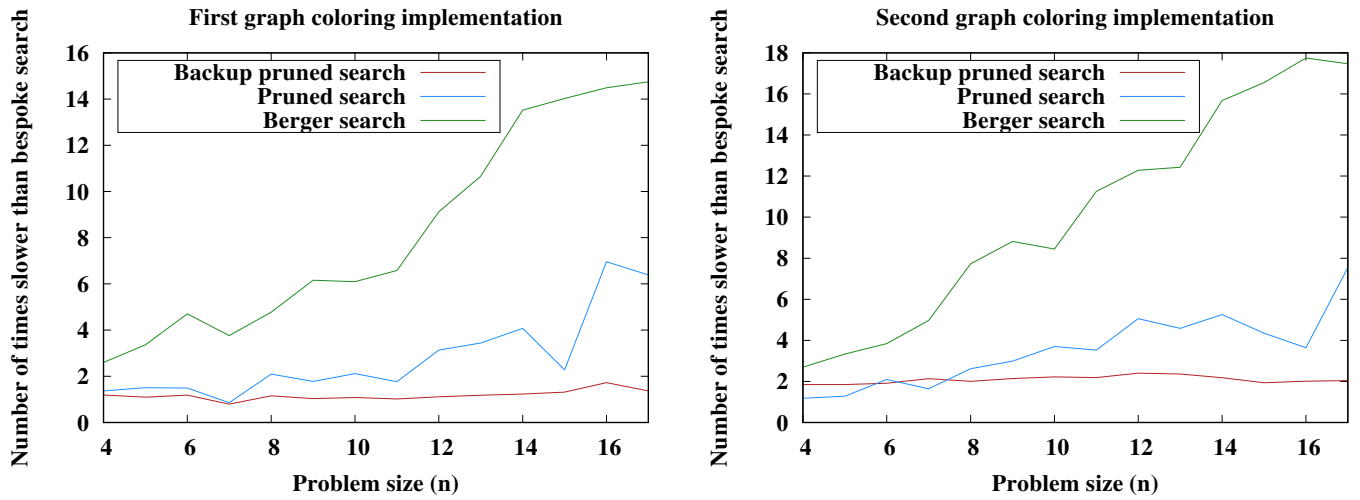


Figure 3.2: The amount of time taken by each search using the first graph colouring implementation, as a multiple of that of the corresponding bespoke search. The example graphs were generated using the Erdős-Rényi model for different values of n , where $p = 0.2$.

Search algorithm performance ratios from finding colourings for Erdős-Rényi graphs, generated with different values for n and $p = 0.5$

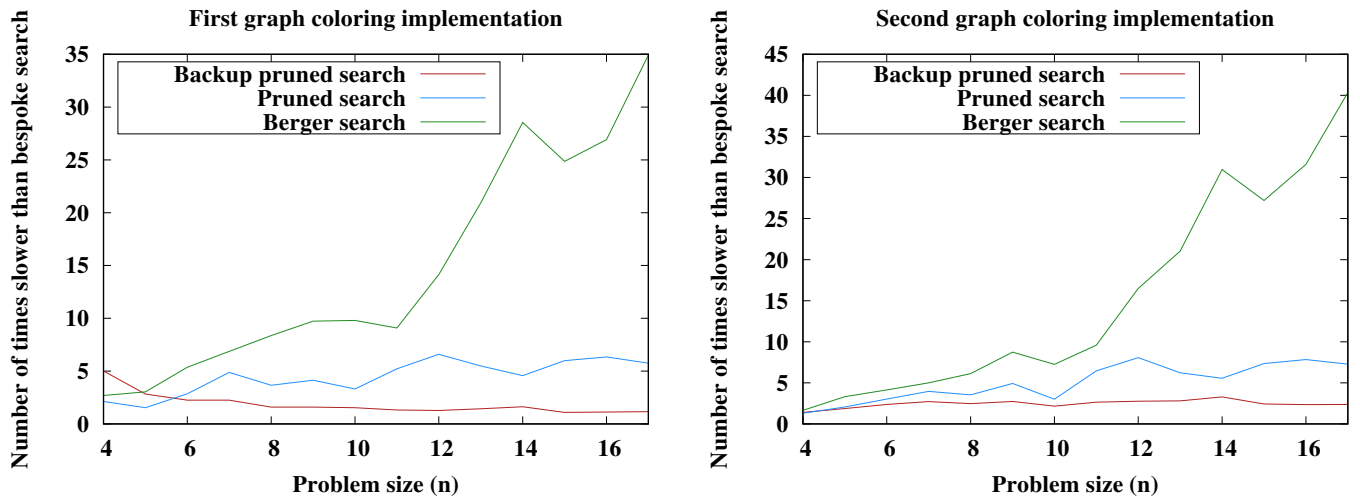


Figure 3.3: The amount of time taken by each search using the first graph colouring implementation, as a multiple of that of the corresponding bespoke search. The example graphs were generated using the Erdős-Rényi model for different values of n , where $p = 0.5$.

Search algorithm performance ratios from finding colourings for Erdős-Rényi graphs, generated with different values for n and $p = 0.8$

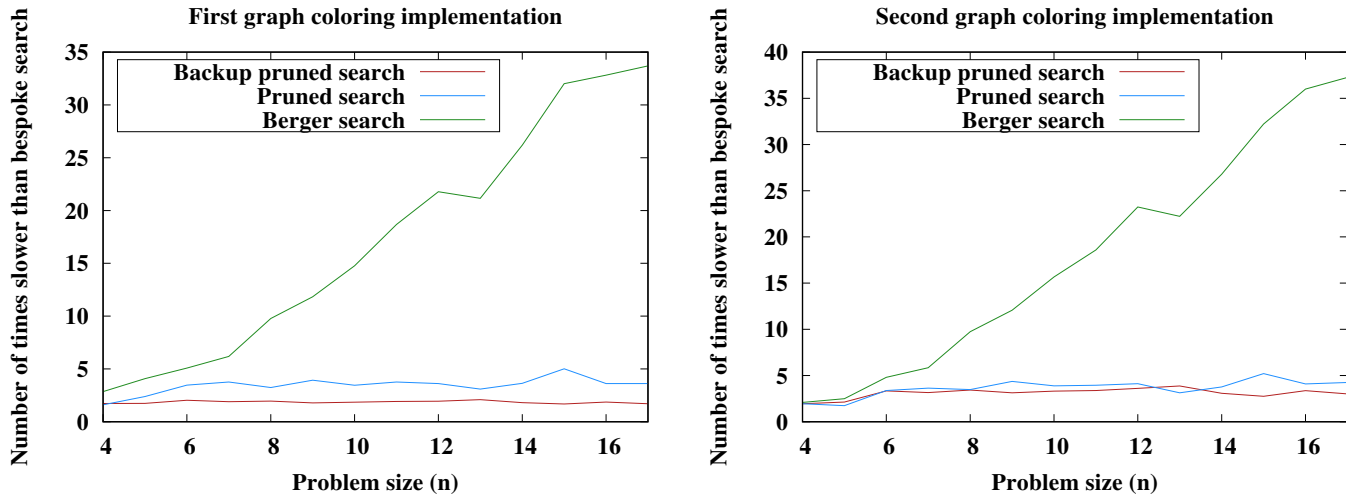


Figure 3.4: The amount of time taken by each search using the first graph colouring implementation, as a multiple of that of the corresponding bespoke search. The example graphs were generated using the Erdős-Rényi model for different values of n , where $p = 0.8$.

n	Berger	Pruned	Backup pruned	Bespoke implementation absolute time (ns)
4	2.09	1.9	1.9	1514
5	2.49	1.7	2.1	1638
6	4.80	3.3	3.3	22961
7	5.84	3.6	3.1	33347
8	9.74	3.4	3.4	22555
9	12.0	4.3	3.1	66262
10	15.6	3.8	3.3	41234
11	18.5	3.9	3.3	37261
12	23.2	4.1	3.6	30615
13	22.2	3.1	3.8	14593
14	26.7	3.7	3.0	34250
15	32.2	5.2	2.7	73840
16	35.9	4.0	3.3	33106
17	37.2	4.2	3.0	45336

Table 3.8: The amount of time taken by each search using the second graph colouring implementation, as a multiple of that of the corresponding bespoke search. The example graphs were generated using the Erdős-Rényi model for different values of n , where $p = 0.8$.

3.3.1 A large caveat

It was discovered that for many of the graphs that were used in these experiments, the pruned and backup pruned search algorithms did not arrive at the same solution as that of the bespoke and Berger search implementations. Additional investigation demonstrated that the pruned and backup pruned searches were beginning some of the searches with σ_0 equal to 1 rather than 0. This investigation is detailed in appendix D.1. Some further analysis by John Longley revealed that this was due to a particular subtlety stemming from the fact that the pruned and backup-pruned search algorithms adapt dynamically to the order in which components are inspected during an evaluation of $P(\sigma)$.

As an example of this, in line 3 of listing 3.3.4, it is checked whether the colour at vertex i is equal to the colour at vertex j . The first case in which this check will be made will be when i is equal to 1, as line 6 prevents this line from being evaluated when i is equal to zero. In the case of a graph such as the example contained in figure 3.5, initially the check at line 6 will cause i to iterate to 1. The dynamic adjustment functionality of the pruned and backup pruned search implementations means that they will both then treat this to be the starting point of the search, rather than when σ_0 was equal to zero, due to the fact that line 3 of listing 3.3.4 evaluates the condition $((p\ i) = (p\ j))$. This resulted in the discrepancy in the returned solutions that was described at the beginning of this section. This phenomenon would not have been observed if line 3 had contained `if ((p j) = (p i))` instead of `if ((p i) = (p j))`. This was verified by modifying the implementations that were used to this effect. It was indeed observed that the pruned and backup pruned search algorithms then obtained the same solutions as the Berger and bespoke implementations.

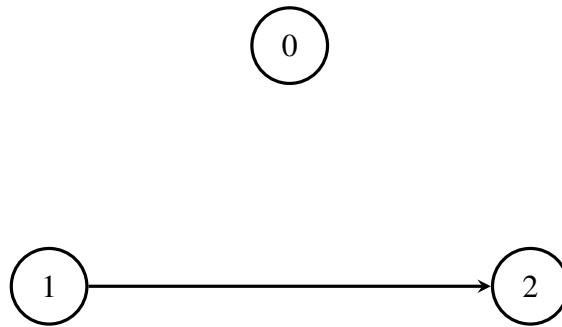


Figure 3.5: An example of a graph that will result in the pruned and backup pruned search implementations beginning their search from a different start point to the bespoke, Berger and naive search implementations, in the case of the `SearchProblem` implementations that were used for these experiments.

In spite of this discrepancy that was found in the solutions, performance comparisons can still be made between the pruned and backup pruned searches, and between the Berger and bespoke searches. Due to the fact that the `property` function in this case affected the order in which S was searched by the pruned and backup pruned search algorithms however (and therefore the heights at which these algorithms could prune

solutions), no meaningful comparisons can be made between either of these two and the Berger or bespoke implementations.

It should also be noted that the discrepancy that has been described in this section should be considered particularly when viewing the results in table 3.2, in which the pruned and backup pruned searches appear to have found solutions faster than the bespoke implementation. This is almost certainly due to the fact that these algorithms begun their search at a different point in S to that of the Bespoke algorithm.

3.3.2 Implementation-dependent performance

All of the results tables containing absolute timings for the bespoke graph colouring algorithms demonstrate a difference in performance between the two graph colouring algorithms that were implemented. In the vast majority of cases, it can be seen that the second implementation was between two and three times faster than the first at finding the first colouring of each input graph. An explanation as to why lies with the implementation of `isEdge`, which uses the adjacency list implementation given in listing 3.3.2. The first graph colouring algorithm that was implemented (given by listing 3.3.3) will evaluate this function almost every time $P(\sigma)$ is evaluated. The second algorithm however will instead check whether two vertices have the same value before before calling the `isEdge` function. This equality check will be considerably less computationally expensive than an evaluation the `isEdge` function, and it will mean that `isEdge` is only used when it is needed. This should mean that for the second implementation, most evaluations of $P(\sigma)$ will be considerably less expensive than those of the first implementation. This spared expense would certainly result in the second algorithm taking less time to find a colouring than the first.

3.3.3 The behaviour of the pruned and backup pruned searches

While the pruned and backup pruned searches may not be compared to the Berger or bespoke searches for the reasons given in section 3.3.1, because they were always found to obtain the same solutions, they may still be compared with each other. Figures 3.2, 3.3 and 3.4 seem to suggest that in some of the example graph colouring problems that were considered, the pruned search algorithm performed more similarly to the backup pruned search algorithm. It can be seen that for both graph colouring algorithm implementations that as p is increased, the ratios of the pruned and backup pruned search become more similar.

Table 3.2 demonstrates that for the larger graphs that were generated using these particular values, there was no valid colouring consisting of four colours. It might be the case that each search algorithm could determine this with very few evaluations of $P(\sigma)$, pruning off branches of S earlier than in problems like N-queens or colouring more sparse graphs. As there would be fewer evaluations of $\mathbb{P}(\sigma)$, the limitation of the pruned search will be less apparent, as it will not be un-necessarily fully evaluating $P(\sigma)$ as much as when there is a colouring to be found. As a result of this, the performance of the pruned and backup pruned search might expected to be more similar, in the case where they are both being used to find colourings of dense graphs with four colours. When $p = 0.5$, it is possible that more backtracking was required to determine that there were no valid colourings, and as a result of this, the pruned search timing

ratios demonstrated their usual approximately linear growth rate. The reasons behind this expected linear growth rate are discussed in section 3.2.1.

3.3.4 Value instability

Figures 3.2, 3.3 and 3.4 all demonstrate the "dips" that were discussed in section 3.2.2. This is likely to be due to an inappropriate number of repeat runs of the pruned and Berger search algorithms. A considerable amount repeats (numbering in the hundreds of thousands) was required to reduce the effects of the noise phenomenon that was described in section 3.1. It could well be the case that there were not enough repeat runs included in the averages for the Berger and pruned searches, and would be expected that their ratio growth curves would become smoother as the number of repeat runs was increased.

3.4 Computing 2-player Nash Equilibria

A "pure Nash equilibrium" is a "profile" of strategies (in which each participating player "plays" a strategy with a probability of 1.0) such that no player will benefit by unilaterally deviating from their strategy. The problem of computing 2-player pure Nash equilibria entails its location within a payoff matrix, the rows of which correspond to one player's possible strategies and the columns correspond to those of the other player. Elements of this matrix are tuples containing the payoffs for each player, given that they have each played a strategy (or selected a row or column in the payoff matrix).

Cast as a generic search problem, σ simply specifies the row to be selected by the row player, and the column to be selected by the column player. In this case, P would be that for the row player, the tuple specified by σ contains a payoff for that player that is the maximum across all columns (given that row), and for the column player, that the tuple contains a payoff for that player that is the maximum across all rows (given that column). Listing 3.4.2 contains an implementation with this behaviour, built on top of the generic search framework that was provided.

Listing 3.4.1:

```
1  type PayoffMatrix = { numPlayers : int,
2                        numRows : int,
3                        numCols : int,
4                        isRowMax : int → int → bool,
5                        isColMax : int → int → bool } ;
```

Listing 3.4.2: The implementations of `isRowMax` and `isColMax` that were used are contained within appendix C.3.

```
1  fun isNE2D pMat : SearchProblem =
2    let fun testStrategy (pMat : PayoffMatrix) (p : int → int) =
3      ((#isColMax pMat (p 0) (p 1))
4       andalso (#isRowMax pMat (p 1) (p 0))) ;
5    in
6      { dimensions = List.tabulate (2, fn i → if i=0 then (#numRows pMat)
```

```

7                                     else (#numCols pMat)) ,
8         property = fn p → testStrategy pMat p }
9     end ;

```

Two small problems were considered. The first was an example of the prisoner's dilemma (in which there is a Nash equilibrium), and the second was a game of rock paper scissors (in which there is no pure Nash equilibrium). Unfortunately, timing constraints did not allow for larger examples (with hundreds or thousands of rows and columns) to be manufactured. The small size of these problems necessitated timings being averaged over 10,000,000 repeats, as nanosecond precision was required for comparison. For reasons to be discussed later in this section, the naive search was also included in this investigation. Figures 3.6 and 3.7 contain the payoff matrices for the prisoner's dilemma and rock paper scissors problems respectively.

It was verified that all search methods produced the same expected result, for both problems that were considered. The code for the bespoke implementation for finding Nash equilibria for two player games is contained in appendix A.4.

P1 \ P2	Cooperate	Defect
Cooperate	0	<u>1</u>
Defect	<u>1</u>	<u>-1</u>

Figure 3.6: The payoff matrix of the prisoner's dilemma game. Values which are underlined correspond to a "best response" for a player, given that the other player has played a certain strategy. Cells with both of their values underlined correspond to pure Nash Equilibria.

P1 \ P2	Rock	Paper	Scissors
Rock	0	<u>1</u>	-1
Paper	<u>1</u>	0	<u>1</u>
Scissors	-1	<u>1</u>	0

Figure 3.7: The payoff matrix of the rock paper scissors game. Values which are underlined correspond to a "best response" for a player, given that the other player has played a certain strategy. In this example, no cell contains two best responses, and thus this is an example of a game in which there are no pure Nash equilibria.

Table 3.9 contains a summary of the results obtained from investigating the performance of each search algorithm on the problems discussed.

Search method	Rock paper scissors	Prisoner's dilemma
Naive search	1.246	0.985
Berger search	1.653	1.285
Pruned search	1.681	0.925
Backup Pruned search	1.974	1.053
Bespoke search absolute time (ns)	1513	1510

Table 3.9: The time spent by each generic search algorithm to find one solution to each game theory problem that was considered, as a multiple of that of the bespoke Nash equilibrium search algorithm that was used.

3.4.1 A search problem that doesn't entail back-tracking

Table 3.9 demonstrates that in the case of computing Nash equilibria, the less sophisticated generic search methods might outperform the pruned and backup pruned search methods. While this might be attributed to the fact that the search problems considered were small in comparison to the others that have been considered so far in this chapter, there is an alternative explanation. The computation of Nash equilibria for games such as rock paper scissors and the prisoner's dilemma is an example of a search problem that can't be pruned. As a result, all possible solutions must be explored until a solution is found. This means that regardless of the search method that is used, all search methods will explore the same branches, and the benefits of backtracking will not be exploited. The simplicity of the naive search may mean that it would have a lower overhead than the pruned and backup pruned approaches in such cases. The results for the prisoner's dilemma would suggest that this particular trend could just be an artefact of the noise phenomenon that was described in section 3.1. It was unfortunately not possible to average much more than 10 million repeats for this experiment due to a limit placed upon the size of `vectors` in `sml`, and so this noise could not be reduced any further. It might at least be concluded that the performance of all of the search algorithms that were considered for this particular investigation was similar in this investigation.

3.5 The completion of partial Latin squares and Sudoku puzzles

Assuming the definition of Latin squares given in chapter 1, a Sudoku puzzle could be considered to be the 9×9 case of Latin squares with an additional constraint. This constraint is that all values within a "block" must also be a permutation, where a block is one of the nine non-overlapping 3×3 sub-regions of a 9×9 grid.

One way of casting the Sudoku and Latin squares completion problem such that they adhere to the type signature given by listing 2.1.1 is that every element in the Sudoku or Latin square grid must be unique in its row and column, and in the case of Sudoku, its "block". Listing 3.5.2 contains an implementations of this property for 9×9

Latin completion. for use with John Longley's generic search framework. The Sudoku implementation that was written and tested is contained in appendix C.4.

Listing 3.5.1:

```
1  type LatinSquare = { numInitVals : int,  
2                        isUniqueInCol : (int → int) → int → bool,  
3                        isUniqueInRow : (int → int) → int → bool,  
4                        isEmpty : int → bool } ;
```

Listing 3.5.2:

```
1  fun isLatinSquare lSquare : SearchProblem =  
2    let fun testElement (lSquare : LatinSquare) (p : int → int) i =  
3      let val test = ((#isUniqueInRow lSquare p i)  
4                    andalso (#isUniqueInCol lSquare p i)) in  
5        test  
6      end  
7    fun testElementsFrom lSquare p i =  
8      i=81 orelse (testElement lSquare p i  
9                andalso testElementsFrom lSquare p (i+1))  
10   in  
11     { dimensions = List.tabulate((81- (#numInitVals lSquare)), fn _ ⇒ 9) ,  
12       property = fn p ⇒ testElementsFrom lSquare p 0 }  
13   end ;
```

The implementations of `isUniqueInRow`, `isUniqueInCol` and for the Sudoku implementation, `isUniqueInBlock` are contained in appendix C.5. Two examples were considered. The first is given by figure 2.1, and the second is given by figure 3.8. As a preliminary exercise, it was verified that each generic search method could find the correct solutions to each of these puzzles.

The amount of time required for each generic search method to find the solution to each puzzle, both as a partial Latin squares completion problem and a Sudoku problem was measured. Table 3.10 the averages of these values, in μ s. Faster implementations were averaged over 100,000 repeats, while slower implementations were instead averaged over 50 repeats.

7		1	5		0		4	6
	0	6			3	1	7	5
	5			6			2	8
1				5	4	8	0	
8	4	3	2	0			1	7
		5	7	1		6	3	
4	1		6	7	5		8	
6	3		0	4		7	5	1
		2	8		1		6	

Figure 3.8: The second 9x9 partial Latin square and Sudoku problem that was considered in these investigations.

Search method	First puzzle (four empty cells), partial Latin square	Second puzzle (thirty empty cells), partial Latin square	First puzzle, Sudoku	Second puzzle, Sudoku
Berger search	576466	58033328	1173	2625555
Pruned search	330	3835666	733	172222
Backup Pruned search	54	215799	115	7400

Table 3.10: The time spent by each generic search algorithm to find the solution to each of the puzzles that were considered, both as a partial Latin squares completion problem and as a Sudoku problem.

3.5.1 Improving performance by adding more constraints

3.10 demonstrates that for small partial Latin square completion puzzles, it might take a pruned and backup pruned search algorithm comparatively less time to find a solution than if they were instead attempting to solve the same 9×9 square as a Sudoku problem. It may be of surprise therefore to see that for larger Latin squares puzzles, that it takes longer to complete the Latin square than it does to solve that square as a Sudoku puzzle instead. This can be attributed to the fact that the additional constraint required by the Sudoku search problem enables for branches in S to be pruned away earlier, resulting in less evaluations of $P(\sigma)$. This therefore demonstrates an example

of a search problem for which adding an additional constraint can speed up the location of a solution by orders of magnitude (which might not come as any particular surprise), as any Sudoku solution is also a complete Latin square.

3.6 Additional comments

It can be seen throughout the examples considered in this chapter, when comparing the generic search implementations to their corresponding bespoke implementations, that the use of a generic search framework has simplified the code that was required for each problem considerably. The number of lines of code for the generic search implementations was considerably less than that of the bespoke implementations.

It is also argued that when inspected, the generic search implementations express their behaviour with considerably more clarity than the bespoke implementations, partially due to the abstraction away from the underlying search machinery. Section 3.3.1 also demonstrated that the pruned and backup pruned search methods demonstrated extra flexibility over that of the bespoke implementations, as it is possible for the programmer to influence the path that the search algorithm takes through S .

The majority of the implementations also demonstrate considerable flexibility with how certain underlying aspects of the problem are implemented. An example of this would be the `isEdge` function that is used in the graph colouring search problem, which is completely agnostic to the representation method of its underlying graph.

The increased flexibility that the generic search framework provides, in conjunction with the clarity with which it enables the programmer to express a search problem certainly makes it an attractive alternative to implementing a problem-specific bespoke search from scratch. In the next chapter, it will be shown how some of the ideas behind this generic search framework can be used for another type of problem.

Chapter 4

Generic search techniques and arbitrary precision real number integration

4.1 A different type of search problem

In the search problems that have been considered so far, S can have arbitrary but finite dimensions and P has been a function of the form $(int \rightarrow int) \rightarrow bool$. That is to say that given a point σ in S , each σ_i may take any integer value, provided that it does not exceed the boundaries of S . Additionally, $P(\sigma)$ will return a value indicating whether or not the point σ satisfied the property P . In [6], John Longley demonstrates how this framework might be adapted for the purposes of real number integration.

4.2 A novel approach to real number integration

Suppose that the dimensions of S are now restricted such that $\forall i, \sigma_i \in \{0, 1\}$, but that S is now infinite. In this case, a point x in S might now be represented by a stream of binary digits, which in `sml` might be represented by the type given in listing 4.2.1. It might be to the reader's surprise that for this problem, a solution is being sought in an infinite search space. There are examples however such as in [4], in which it is demonstrated that certain types of infinite set can be searched exhaustively in a finite amount of time.

Listing 4.2.1: This `stream` type consists of a head and a tail (which can be pattern matched). The head is the current element in the stream, and the tail contains all of the necessary information to access the rest of the stream. Given an input of type `unit`, it will return a new stream with a head equal to the next element in the previous stream, and another tail, which can in turn be used to access later elements in the stream.

```
1 datatype 'a stream = Cons of 'a * (unit → 'a stream) ;
```

The property of this search problem is now a real-valued function f defined on S , where points on S are interpreted as binary values within the unit interval. This function could certainly be represented by $(int \text{ stream} \rightarrow int \text{ stream})$, however for the purpose of computing $f(x)$ to some desired precision 2^{-k} , only k digits of the output stream are

required. Because of this, f will instead be represented as $(\text{int stream} \rightarrow \text{int})$, where the integer returned will satisfy the inequality $|f(x) - \frac{P}{2^k}| < 2^k$. To get $\int_{x=0}^1 f(x) dx$ to a precision within 2^{-k} , it suffices to know every sampled point of f to within an error of 2^{-k} . It should be noted that typical approaches to exact real number computation (such as in [12]) use the digits $\{-1, 0, 1\}$, however because f will return an integer rather than an `int stream`, only 0 and 1 will be required for simple experiments.

4.3 Performance and accuracy considerations

Assuming that f is a one-variable function, an algorithm conducting the search described in section 4.2 is effectively evaluating different points on f with a variable sampling width. This sampling width will always be as large as possible, bound by the constraint that computed values must be within a precision of 2^{-k} . This has two important consequences. The first is that for relatively flat regions in a given f , this integration method might only need to take a small number of samples. In contrast, many traditional methods of numerical integration require that the user specify a fixed sampling width. In such cases, certain f may then be sampled thousands of times, completely un-necessarily. Provided that there is not too much of an overhead associated with evaluation of $f(x)$ with this new integration method, it may offer a significant improvement in performance over that of some of its alternatives.

Parameterising the precision of an output integral enables guarantees to be made about the corresponding error in its calculation. This is again in contrast with integration methods that make use of a fixed sampling width, because the specification of a sampling width does not allow for any guarantees about error to be made. In the case of functions that exhibit highly chaotic behaviour for example, there might be many oscillations between two sample points that will not be accounted for, resulting in aliasing. The advantage of a guaranteed error bound becomes particularly apparent in such cases, as a value for k can be specified such that this aliasing is either reduced or entirely avoided.

4.4 Implementations

4.4.1 A logging integrator

As it was suggested in sections 4.1 and 4.2, the integration problem that has been described so far can be considered analogous to a search problem, except that S now the unit interval and may be represented as a binary tree.

In order to compute the integral of an input function f to the required precision, an implementation of the described integration method needs to know both the value returned from the evaluation of $f(x)$, and how many digits of the input stream were used to arrive at this value. Listing 4.4.1 gives a function that will wrap an `int stream x` such that a log is kept of the digits of x that were used to return a value. With this information, an implementation can then calculate $\frac{P}{2^k}$ and then evaluate the input function g at the next value for x . For example, if x was a stream of zeroes and $f(x)$ used the first two digits of x , the next x at which to evaluate f would be 01.

Listing 4.4.1: `logWrap` behaves as a thunk. Given an input stream, it will return that stream, however it will have the additional side effect of updating a `ref` with the new binary digit that has been requested.

```

1  fun updateLogs i (Cons(hd,tl)) iLog xLog =
2    if i = (!iLog)+1 then
3      (iLog:=i; xLog := hd::(!xLog))
4    else () ;
5
6  fun logWrap i (Cons(hd,tl)) iLog xLog =
7    let val x as (Cons(hd',tl')) = tl() in
8      Cons(hd, fn () => (updateLogs i (x) iLog xLog;
9        logWrap (i+1) x iLog xLog))
10   end ;

```

An integrator exploiting these functions for the purposes outlined above was implemented. The input function of type $(int\ stream \rightarrow int)$ is evaluated by this implementation at a point represented by the wrapped binary stream x for different values of x in the unit interval. The fact that this is guaranteed to terminate for all total $g : int\ stream \rightarrow int$ is a consequence of König’s lemma. This implementation is included in appendix B.1.

4.4.2 A limitation of the logging integrator

Similarly to the limitation with the pruned generic search approach detailed in chapter 2, the thunk given in listing 4.4.1 means that there is scope for un-necessary re-computation.

Recall that in section 4.2, it was stated that the number of digits that is returned by $f(x)$ will be the number that was required for it to return a value. The digits of x that were used are used by the implementation to determine the next point from which the integration must begin (as these digits correspond to a value that is equal to the width of the sample). Because of this, most of the digits of x that were evaluated at the previous point are present in the next. The thunk given in listing 4.4.1 will always evaluate the stream that it wraps from its first element in spite of this. This means that for computationally expensive input functions, this integrator will spend considerably more time integrating that function than it should really need to. Building upon some of the ideas presented in chapter 2, section 4.4.3 will describe the implementation of an integrator that does not suffer from this limitation.

4.4.3 An integrator that makes use of continuations

An integrator was implemented that is capable of using ”resumable exceptions” (as described in chapter 2), in order to overcome the limitation of the logging integrator that is detailed in section 4.4.2. It was written as a modification of the backup pruned search implementation by John Longley [5], also described in chapter 2. The modifications that were required will be included in this section.

The `resumption` type that was used in the backup pruned search algorithm was of the type $(int \rightarrow int) \rightarrow bool$. This reflected the fact that $P(\sigma)$ was being evaluated for

a given point in S , represented by a function of type $(int \rightarrow int)$, where P returned a boolean value indicating whether the sought property was present at the position σ . This integration problem considered a function of the form $(int\ stream \rightarrow int) \rightarrow int$, and so the resumption type was amended in order to accommodate this.

A function of type $(int \rightarrow int) \rightarrow int\ stream$ (given by listing 4.4.2) was also required, as the `catchcont` did not directly accommodate the $int\ stream$ type, but the function to be integrated required an `int stream` as an input.

Listing 4.4.2:

```
1  fun funToStream f = funToStream' f 0
2  and funToStream' f i = Cons(f i, fn () => funToStream' f (i+1)) ;
```

The integrator function also accumulated a value (the sum of all evaluations of $f(x)$ so far), as opposed to providing a list of solutions (as the backup pruned search implementation did). Finally, the backup pruned search implementation allowed for the values of a given σ to be any integer. In the case of this integration problem however, the digits of x must either be a zero or a one. This restriction was therefore enforced by the integrator. The full source code for this implementation is contained in appendix B.2.

Similarly to the way in which the backup pruned search could resume an evaluation of $P(\sigma)$, this implementation was capable of starting the evaluation $f(x)$ from the next digit of x that is required, rather than from the beginning of x .

4.5 A brief correctness check and performance comparison

The correctness of each implementation was verified using the function given in listing 4.5.1. This function was evaluated 1,000,000 times by each integrator and the resultant times averaged over these repeats, in order to evaluate their performance.

Listing 4.5.1: This function will return the index of the first 1 that it finds in an input stream, up to an index value of 28. Integrated over the unit interval, h should be equal to the following sum: $\sum_{i=0}^{28} \frac{i}{2^i}$, equal to a value of 1.99999988824 to `sml` real precision.

```
1  fun h x =
2    let fun h' i (Cons(hd,t1)) =
3          if i = 28 then i
4          else if hd = 1 then i
5          else h' (i+1) (t1())
6    in
7      h' 0 x
8    end ;
```

Both integrators produced the expected value of 1.99999988824. The average time taken for the logging integrator to evaluate h was 38,381 μ s, while the integrator that used continuations averaged 17,427 μ s on the same function. It might be expected for

the continuation integrator to have taken considerably less than half of the time taken by the logging integrator, however it should be noted that h was not a particularly computationally expensive function. Throughout the examples that were considered in chapter 3, it was seen that for computationally inexpensive problems (such as the colouring of a sparse graph, or small 2-player Nash equilibrium problems), the performance of the pruned and backup pruned searches could be very similar. The performance benefits of the continuation integrator might therefore only emerge when more computationally expensive functions are used.

Chapter 5

Concluding remarks

5.1 A brief summary

The overhead of a pruned search will linearly with the size of the search problem that it is attempting to solve, due to the fact that it will always evaluate P from the first index of σ . In comparison, the backup pruned approach will only evaluate P from the index of σ that is actually required, and as a result of this its overhead remains approximately constant as the size of a problem will increase.

Generic search algorithms might sometimes perform more favourably for problems in which it takes less time to find a solution, due to the fact that in simple search problems, search algorithms will on average traverse S to a lesser depth. As a result of this, the algorithms will exhibit less of their computationally expensive backtracking behaviour than they might usually before they find a solution.

The overhead of the Berger search implementation will scale approximately quadratically as a consequence of the way in which it must be implemented in `sml` in order for it to behave functionally.

There would appear to be a larger performance overhead when a pruned generic search algorithm is implemented in C++ or Java, than when it is implemented in `sml`. It is conjectured that this is due to the features available in each language, however it is also possible that there was un-necessary computation being carried out by the C++ implementation, which demonstrated an overhead growth rate comparable to that of the Berger search implementation.

There are search problems for which small implementation differences can have an effect upon the path that a search algorithm might take through the corresponding S , and even the position from which it will start. Additionally, Backtracking is not always possible in some search problems, and as a result, generic search algorithms that would otherwise not be competitive can perform comparably to more sophisticated implementations. Adding additional constraints to a search problem can also speed it up considerably, due to the fact that pruning will occur earlier in the search algorithms traversal of S .

It has been repeatedly seen throughout the experiments that were conducted that the use of a generic search framework has simplified many of the problems that were investigated considerably. Its use also offered enhanced code clarity, as well as flexibility in multiple respects (both in terms of how the programmer is sometimes able to influence the path with which a search algorithm traverses S , and the way in which many components of the problem can be completely implementation agnostic). This expressive power, in conjunction with the competitive performance of the backup pruned generic search algorithm for some problems might mean that a generic search framework would be an attractive option in certain scenarios.

Finally, generic search techniques can be applied for the purpose of arbitrary precision real number integration. This approach potentially offers performance benefits through the use of dynamically specified sampling widths, as well as a guaranteed error bound that can be parametrised and specified by a user. In particular, it might have applications integrating particularly chaotic functions for this reason. The integration approach that was made use of continuations did not offer as much of a performance increase over the other integration approach that relied upon logging as was expected, however this might just be due to the fact both implementations were only tested on a relatively computationally inexpensive function.

5.2 Further work

Section 3.2.2 described a possible explanation for the possibly quadratic growth in the overhead of the Berger search implementation when problem size is increased linearly. It was suggested that by instead using mutable arrays in conjunction with array slices, the overhead might instead be expected to grow linearly. The performance of such an implementation could then be compared to that of the purely functional Berger search implementation, so that the overhead of its functional behaviour might be quantified. The Berger search implementation can also be modified so that it can find all solutions. This might be achieved by making the algorithm resume from the point in S where it last located a solution, for example. This would enable for many of the experiments that were carried out in these investigations to be carried out with each algorithm finding all solutions instead of just one, enabling for several of the conclusions that were drawn in chapter 3 to be investigated further.

Graph colouring is one of Karp's 21 NP-complete problems [2]. For such problems, there are no known ways of efficiently locating P within S , and so a generic search may offer comparable performance to an algorithm that has been designed specifically for finding the colourings of a graph. It might therefore be of interest to investigate how competitive a generic search algorithm would be for such a problem.

There are examples of NP-complete graph theory problems such as finding Hamiltonian circuits, for which the backup pruned search algorithm might offer competitive performance. Such problems frequently entail the location of permutations within S . It might therefore be of interest to try and quantify the overhead associated with the use of a generic search algorithm (which will not be capable of taking advantage of the fact that permutations are being searched for) in such cases. The listing in appendix C.6 gives a function inspired by Heap's algorithm. In a constant amount of time, and

requiring a linear amount of memory, this function is capable of cycling permutations without repeating any previously explored permutations. An implementation capable of finding Hamiltonian circuits using John Longley's generic search framework that was tested to behave correctly is given in appendix C.7.

The integrators that were implemented were only tested on a relatively computationally inexpensive function. A more rigorous performance comparison could be made using computationally expensive functions (for example, integrating x^2), so that the speculated performance benefits of the continuation-based integrator can be investigated.

Bibliography

- [1] T. Easton and R. G. Parker, “On completing latin squares,” *Discrete Applied Mathematics*, vol. 113, no. 23, pp. 167 – 181, 2001.
- [2] R. M. Karp, *Reducibility among Combinatorial Problems*, pp. 85–103. Boston, MA: Springer US, 1972.
- [3] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou, “The complexity of searching a graph,” *J. ACM*, vol. 35, pp. 18–44, Jan. 1988.
- [4] M. H. Escardó, “Infinite sets that admit fast exhaustive search.,” in *LICS*, pp. 443–452, Citeseer, 2007.
- [5] J. Longley, “Experiments in generic search algorithms.” SML of New Jersey source file (private communication).
- [6] J. Longley, “When is a functional program not a functional program?,” *SIGPLAN Not.*, vol. 34, pp. 1–7, Sept. 1999.
- [7] U. Berger, *Totale Objekte und Mengen in der Bereichstheorie. PhD Thesis, Ludwig Maximilians-Universität, München.* na, 1990.
- [8] “The vector structure.” <http://www.sml-family-irg/Basis/vector.html#SIG:VECTOR.update:VAL>, 2004. Accessed: 18.08.2016.
- [9] K. Xia, *Efficient generic search using Java and C++.* na, 2016.
- [10] G. Gonthier, “Formal proof—the four-color theorem,” *Notices of the AMS*, vol. 55, no. 11, pp. 1382–1393, 2008.
- [11] A. R. Paul Erdős, “On random graphs,” *Publ. Math. Debrecen*, vol. 6, pp. 290–297, 1959.
- [12] A. Bauer, M. H. Escardó, and A. Simpson, “Comparing functional paradigms for exact real-number computation,” in *International Colloquium on Automata, Languages and Programming*, pp. 488–500, Springer, 2002.

Appendix A

Bespoke search problem implementations

A.1 N-queens

```
1  structure BespokeQueens = struct
2
3  local
4
5  fun testQueenPair a i j =
6      let val (jpos,ipos) = (Array.sub(a,j), Array.sub(a,i)) in
7          not (ipos = jpos orelse ipos-jpos = i-j orelse ipos-jpos = j-i)
8      end ;
9
10 fun testQueenPairsFrom a i j =
11     j=i orelse (testQueenPair a i j
12         andalso testQueenPairsFrom a i (j+1)) ;
13
14 exception Found of int Array.array ;
15
16 fun arrayToList a = Array.foldr (fn (x,l) => x::l) [] a ;
17
18 fun searchAllExtensions n a i =
19     if i = n then raise Found a
20     else searchAllExtensionsFrom n a i 0
21 and searchAllExtensionsFrom n a i j =
22     if j = n then () else
23     (Array.update(a,i,j) ;
24     if testQueenPairsFrom a i 0
25     then searchAllExtensions n a (i+1) else () ;
26     searchAllExtensionsFrom n a i (j+1)) ;
27
28 fun searchAllExtensions' acc n a i =
```

```
29     if i = n then arrayToList a :: acc
30     else searchAllExtensionsFrom' acc n a i 0
31 and searchAllExtensionsFrom' acc n a i j =
32     if j = n then acc else
33     (Array.update(a,i,j) ;
34      let val acc' =
35          if testQueenPairsFrom a i 0
36          then searchAllExtensionsFrom' acc n a (i+1)
37          else acc
38      in searchAllExtensionsFrom' acc' n a i (j+1)
39      end) ;
40
41 in
42
43 fun findOne n =
44     (searchAllExtensions n (Array.array(n,0)) 0 ; NONE)
45     handle Found a => SOME (arrayToList a) ;
46
47 fun findAll n = List.rev (searchAllExtensions' [] n (Array.array(n,0)) 0)
48
49 end
50
51 end ;
```

A.2 Graph colouring

```
1  structure BespokeGraColoring = struct
2
3  local
4
5  fun testColoring (g : Graph) p i j =
6      let val (pi,pj) = (Array.sub(p,i), Array.sub(p,j)) in
7          if (#isEdge g i j) then not (pi = pj) else true
8      end ;
9
10 fun testVertexPairsFrom g p i j =
11     j=i orelse (testColoring g p i j
12                andalso testVertexPairsFrom g p i (j+1)) ;
13
14 exception Found of int Array.array ;
15
16 fun searchAllExtensions g k p i =
17     if i = (#size g) then raise Found p
18     else searchAllExtensionsFrom g k p i 0
19 and searchAllExtensionsFrom g k p i j =
20     if j = k then () else
21     (Array.update(p,i,j) ;
```

```
22     if testVertexPairsFrom g p i 0
23     then searchAllExtensions g k p (i+1) else () ;
24     searchAllExtensionsFrom g k p i (j+1)) ;
25
26 fun searchAllExtensions' acc g k p i =
27   if i = (#size g) then arrayToList p :: acc
28   else searchAllExtensionsFrom' acc g k p i 0
29 and searchAllExtensionsFrom' acc g k p i j =
30   if j = k then acc else
31   (Array.update(p,i,j) ;
32    let val acc' =
33      if testVertexPairsFrom g p i 0
34      then searchAllExtensions' acc g k p (i+1)
35      else acc
36    in searchAllExtensionsFrom' acc' g k p i (j+1)
37    end) ;
38
39 in
40
41 fun findOne k g =
42   (searchAllExtensions g k (Array.array((#size g),0)) 0 ; NONE)
43   handle Found p => SOME (arrayToList p) ;
44
45 fun findAll k g = List.rev (searchAllExtensions'
46                             [] g k (Array.array((#size g),0)) 0)
47
48 end
49
50 end ;
```

A.3 Graph colouring alternative implementation

```
1  structure BespokeGraColoringAlt = struct
2
3  local
4
5  fun testColoring (g : Graph) p i j =
6    let val (pi,pj) = (Array.sub(p,i), Array.sub(p,j)) in
7      not (if (pi = pj) then (#isEdge g i j) else false)
8    end ;
9
10 fun testVertexPairsFrom g p i j =
11   j=i orelse (testColoring g p i j
12              andalso testVertexPairsFrom g p i (j+1)) ;
13
14 exception Found of int Array.array ;
15
```

```
16 fun searchAllExtensions g k p i =
17   if i = (#size g) then raise Found p
18   else searchAllExtensionsFrom g k p i 0
19 and searchAllExtensionsFrom g k p i j =
20   if j = k then () else
21   (Array.update(p,i,j) ;
22    if testVertexPairsFrom g p i 0
23    then searchAllExtensions g k p (i+1) else () ;
24    searchAllExtensionsFrom g k p i (j+1)) ;
25
26 fun searchAllExtensions' acc g k p i =
27   if i = (#size g) then arrayToList p :: acc
28   else searchAllExtensionsFrom' acc g k p i 0
29 and searchAllExtensionsFrom' acc g k p i j =
30   if j = k then acc else
31   (Array.update(p,i,j) ;
32    let val acc' =
33      if testVertexPairsFrom g p i 0
34      then searchAllExtensions' acc g k p (i+1)
35      else acc
36    in searchAllExtensionsFrom' acc' g k p i (j+1)
37    end) ;
38
39 in
40
41 fun findOne k g =
42   (searchAllExtensions g k (Array.array((#size g),0)) 0 ; NONE)
43   handle Found p => SOME (arrayToList p) ;
44
45 fun findAll k g = List.rev (searchAllExtensions'
46                             [] g k (Array.array((#size g),0)) 0)
47
48 end
49
50 end ;
```

A.4 Nash equilibrium for 2 players, bespoke implementation

```
1 structure BspokeNE2D = struct
2
3   local
4
5   fun testStrategy (pMat : PayoffMatrix) p =
6     let val (p0,p1) = (Array.sub(p,0), Array.sub(p,1)) in
7       ((#isColMax pMat p0 p1) andalso (#isRowMax pMat p1 p0))
```

```

8     end ;
9
10    exception Found of int Array.array ;
11
12    fun searchAllExtensions pMat p i =
13      if i = 2 then raise Found p
14      else searchAllExtensionsFrom pMat p i 0
15    and searchAllExtensionsFrom pMat p i j =
16      if j = (if i=0 then (#numRows pMat) else (#numCols pMat)) then () else
17        (Array.update(p,i,j) ;
18         if testStrategy pMat p
19         then searchAllExtensions pMat p (i+1) else () ;
20         searchAllExtensionsFrom pMat p i (j+1)) ;
21
22    fun searchAllExtensions' acc pMat p i =
23      if i = 2 then arrayToList p :: acc
24      else searchAllExtensionsFrom' acc pMat p i 0
25    and searchAllExtensionsFrom' acc pMat p i j =
26      if j = (if i=0 then (#numRows pMat) else (#numCols pMat)) then acc else
27        (Array.update(p,i,j) ;
28         let val acc' =
29           if testStrategy pMat p
30           then searchAllExtensions' acc pMat p (i+1)
31           else acc
32         in searchAllExtensionsFrom' acc' pMat p i (j+1)
33         end) ;
34
35    in
36
37    fun findOne pMat =
38      (searchAllExtensions pMat (Array.array(2,0)) 0 ; NONE)
39      handle Found p => SOME (arrayToList p) ;
40
41    fun findAll pMat = List.rev
42      (searchAllExtensions' [] pMat (Array.array(2,0)) 0)
43    end
44
45    end ;

```


Appendix B

Real number integrator implementations

B.1 Logging integrator implementation

```
1  structure LogIntegrator : APRNI = struct
2    local
3      fun incList [] = []
4        | incList (0::tl) = 1::(0::tl)
5        | incList (1::tl) = incList tl
6
7      fun updateLogs i (Cons(hd,tl)) iLog xLog =
8        if i = (!iLog)+1 then
9          (iLog:=i; xLog := hd::(!xLog))
10         else () ;
11
12      fun logWrap i (Cons(hd,tl)) iLog xLog =
13        let val x as (Cons(hd',tl')) = tl() in
14          Cons(hd, fn () => (updateLogs i (x) iLog xLog;
15            logWrap (i+1) x iLog xLog))
16        end ;
17
18      fun evalWithLog (g : int stream -> int) x iLog xLog =
19        let val n = (g (logWrap 1 (x) iLog xLog) )
20          val k = (!iLog)
21        in
22          (xLog := incList (!xLog);
23            (Real.fromInt(n))/(Real.fromInt(Int.fromLarge(exp2 k))))
24        end ;
25
26      fun nextPoint g (ref (hd::tl)) =
27        let val iLog' = ref 0
28          val xLog' = ref []
```

```
29         val x'      = (listToStream (List.rev(hd::tl)) 0)
30     in
31         ((evalWithLog g x' iLog' xLog') + (nextPoint g xLog'))
32     end
33     | nextPoint g (ref ([])) = 0.0 ;
34
35 in
36     fun integrate g =
37         let val x = arrayToStream (Array.fromList([])) 0
38             val iLog = ref 1
39             val xLog = ref []
40         in
41             (evalWithLog g x iLog xLog) + nextPoint g xLog
42         end ;
43     end
44 end ;
```

B.2 Continuation integrator implementation

```
1  structure ContIntegrator : APRNI = struct
2      local
3          fun funToStream f = funToStream' f 0
4          and funToStream' f i = Cons(f i, fn () => funToStream' f (i+1)) ;
5
6          type resumption = (int->int) -> (int*unit) ;
7
8          val restartStack = ref ([] : (int*int*(int->resumption)) list) ;
9
10         exception RestartStackEmpty ;
11
12         fun currentRestart () =
13             List.hd (!restartStack) handle List.Empty => raise RestartStackEmpty ;
14
15         fun pushRestart restart = (restartStack := restart :: (!restartStack)) ;
16
17         fun popRestart () =
18             case !restartStack of
19                 (hd::tl) => ((restartStack := tl) ; hd)
20                 | [] => raise RestartStackEmpty ;
21
22         fun evalWithCont (res : resumption) (markedPoint : (int*bool) array) =
23             case Catchcont.catchcont (fn f =>
24                 res (fn k => (case Array.sub(markedPoint,k) of
25                     (n,true) => n
26                     | (_,false) => f(k)
27                 )))
28             of inl answer => (#result answer,
```

```
29                                     (List.length(!restartStack)),markedPoint)
30     | inr cont =>
31         let val k = #arg cont
32         in
33             Array.update(markedPoint,k,(0,true)) ;
34             pushRestart (k,0,#resume cont) ;
35             evalWithCont (#resume cont 0) markedPoint
36         end ;
37
38     fun increment markedPoint =
39         (increment' markedPoint ; true)
40         handle RestartStackEmpty => false
41     and increment' markedPoint =
42         let val (i,j,res) = popRestart() in
43             if j < 2
44             then (pushRestart (i,j+1,res) ;
45                  Array.update(markedPoint,i,(j+1,true)) ;
46                  markedPoint)
47             else (Array.update(markedPoint,i,(0,false)) ;
48                  increment' markedPoint)
49         end ;
50
51     fun nextPoint' acc res markedPoint =
52         let val (n,k,markedPoint') = evalWithCont res markedPoint in
53             let val acc' = acc
54                 + (Real.fromInt(n))
55                 / (Real.fromInt(Int.fromLarge(exp2 k))) in
56                 if increment markedPoint' then
57                     let val (i,j,restart) = popRestart() in
58                         nextPoint' acc' (restart j) markedPoint'
59                     end
60                 else acc'
61             end
62         end ;
63
64     fun nextPoint res markedPoint =
65         let val (_,_,markedPoint') = evalWithCont res markedPoint in
66             if increment markedPoint' then
67                 let val (_,j,restart) = popRestart() in
68                     nextPoint' 0.0 (restart j) markedPoint'
69                 end
70             else 0.0
71         end ;
72     in
73         fun integrate g =
74             (restartStack := [] ; (* *))
```

```
75         getP (fn p => (g (funToStream p), ()))  
76             (Array.array(40, (0,false))))  
77     end  
78 end ;
```

Appendix C

Additional code

C.1 Berger search implementation that was used for the investigations in chapter 3.

```
1  structure BergerSearch : GENERIC_SEARCH = struct
2
3      (* 'Functional' version with vectors: no in-place update *)
4
5      open Vector ;
6
7      (* Memoization operation, to achieve effect of call-by-need.
8         Observationally equivalent to identity, so morally functional. *)
9
10     fun memoize (thunk : unit -> 'a) =
11         let val cache = ref (NONE : 'a option) in
12         fn () => case !cache of
13             SOME x => x
14             | NONE => let val x = thunk() in
15                         (cache := SOME x) ; x
16         end
17     end ;
18
19     (* Berger's algorithm: selection function for a property P *)
20
21     fun select P allDims =
22         let val lgth = List.length allDims ;
23         fun bestShot fullSol [] _ = fullSol
24           | bestShot partSol (d::dims) i =
25             let val partSol' = concat [partSol, fromList[i]]
26             in if i = d-1
27                then bestShot partSol' dims 0
28                else let val f = bestFun partSol' dims
```

```
29         in if P f then tabulate (lgth, f)
30         else bestShot partSol (d::dims) (i+1)
31     end
32 end
33 and bestFun partSol dims =
34     let val fallback = memoize (fn () => bestShot partSol dims 0)
35     in fn i => if i < length partSol then sub (partSol, i)
36         else sub (fallback(), i)
37     end
38     in bestShot (fromList []) allDims 0
39 end ;
40
41 end (* structure BergerSearch *) ;
```

C.2 Erdős-Rényi graph generation code

```
1 fun erdosRenyiAdjList n p =
2     (* Generate a random number using a seed based off the system time: *)
3     let val rand = (Random.rand (
4         IntInf.toInt(
5             (Time.toSeconds (Time.now())) mod 1000),
6             IntInf.toInt((Time.toSeconds (Time.now())) mod 1000)))
7     in
8         (* Generate a bool list list, indicating vertices
9            in each entry of an adjacency list: *)
10        let val boolList = List.tabulate(n, fn _ =>
11            List.tabulate(n, fn _ =>
12                if ((Random.randReal rand) < p)
13                then true
14                else false))
15            val itList = List.tabulate(n, fn i => i)
16        in
17            (* Map the bool list list onto a corresponding int list list: *)
18            List.map (fn i =>
19                List.filter (fn j =>
20                    List.nth(List.nth(boolList, j), i))
21                    itList)
22                itList
23        end
24    end;
25
26
27 fun newGraphErdosRenyiAdjList n p =
28     size = n, isEdge = isEdgeAdjList (erdosRenyiAdjList n p)
```

C.3 Nash equilibrium 2D isRowMax and isColMax implementations

```
1 fun isColMaxAll (pOffArr2 : int vector Array2.array) =
2   fn row =>
3     fn col =>
4       List.all (fn i =>
5         Vector.sub((Array2.sub(pOffArr2, row, col)),1)
6         >= (Vector.sub((Array2.sub(pOffArr2, row, i)),1)))
7       (List.tabulate((Array2.nRows pOffArr2), fn i => i)) ;
8
9 fun isRowMaxAll (pOffArr2 : int vector Array2.array) =
10  fn col =>
11    fn row =>
12      List.all (fn i =>
13        (Vector.sub((Array2.sub(pOffArr2, row, col)),0)
14        >= (Vector.sub((Array2.sub(pOffArr2, i, col)),0)))
15      (List.tabulate((Array2.nCols pOffArr2), fn i => i)) ;
16
17 fun newPMatArray2 pOffArr2: PayoffMatrix =
18   numPlayers = 2 ,
19   numRows    = Array2.nRows pOffArr2 ,
20   numCols    = Array2.nCols pOffArr2 ,
21   isColMax   = isColMaxAll pOffArr2 ,
22   isRowMax   = isRowMaxAll pOffArr2 ;
```

C.4 Sudoku generic search code

```
1 type Sudoku =      numInitVals : int,
2                   isUniqueInCol : (int -> int) -> int -> bool,
3                   isUniqueInRow : (int -> int) -> int -> bool,
4                   isUniqueInBlock : (int -> int) -> int -> bool,
5                   isEmpty : int -> bool
6
7 fun isSudokuSoln sudoku : SearchProblem =
8   let fun testElement (sudoku : Sudoku) (p : int->int) i =
9       ((#isUniqueInRow sudoku p i) andalso (#isUniqueInCol sudoku p i)
10        andalso (#isUniqueInBlock sudoku p i))
11
12       fun testElementsFrom sudoku p i =
13         i=81 orelse
14         (testElement sudoku p i andalso testElementsFrom sudoku p (i+1)) ;
15   in
16     { dimensions = List.tabulate((81-(#numInitVals sudoku)), fn _ => 9) ,
17       property = fn p => testElementsFrom sudoku p 0 }
18   end ;
```

C.5 Sudoku and Latin Squares element uniqueness check code

```
1  (* Returns a vector that returns how many clues were
2     initially provided before a given index in the flat
3     representation of the puzzle.  *)
4  fun getCulVecFrom initVals =
5      Vector.map(fn k =>
6          (List.length(
7              List.filter (
8                  fn l =>
9                      (not (Vector.sub(initVals,l)=NONE)))
10                     (List.tabulate(k, fn l => l))))))
11      (Vector.tabulate(81,fn k => k))
12
13  (* Given a vector of appropriate indexes,
14     returns a flat vector of indexes which contains
15     the current solution so far (p), as well as the
16     clues that were initially provided.  *)
17
18  fun getVec initVals culVec p itVec =
19      Vector.map (fn k => if Vector.sub(initVals,k) = NONE then
20                          p(k-(Vector.sub(culVec,k)))
21                          else
22                              getOpt((Vector.sub(initVals,k)),0))
23      itVec
24
25  (* Returns a vector containing a column out of the puzzle,
26     given a particular index.  *)
27  fun colVec initVals culVec p i =
28      let val itVec = Vector.tabulate( ((i div 9)+1),
29                                          fn k => (i mod 9)+(9*k))
30      in
31          getVec initVals culVec p itVec
32      end ;
33
34  (* Returns a vector containing a row out of the puzzle,
35     given a particular index.  *)
36  fun rowVec initVals culVec p i =
37      let val itVec = Vector.tabulate( ((i mod 9)+1),
38                                          fn k => (i-(i mod 9)+k))
39      in
40          getVec initVals culVec p itVec
41      end ;
42
43  (* Given a vector and an input index value, checks whether
```

```
44     the key at said index is unique. *)
45 fun isUniqueInVec inVec i =
46     Vector.all (fn k => if (not (k=i)) then
47         not
48             (Vector.sub(inVec,k)=Vector.sub(inVec,i))
49         else true)
50     (Vector.tabulate( (Vector.length(inVec)),
51         fn k => k))
52
53 fun isUniqueInColVec initVals culVec =
54     fn p => fn i =>
55         let val col = colVec initVals culVec p i in
56             isUniqueInVec col (Vector.length(col)-1)
57         end ;
58
59 fun isUniqueInRowVec initVals culVec =
60     fn p => fn i =>
61         let val row = rowVec initVals culVec p i in
62             isUniqueInVec row (Vector.length(row)-1)
63         end ;
64
65 (* Given an element of a 9x9 square
66    (assuming row major indexing), returns the
67    indices of the "block" that contains that element.
68    Unfortunately, this was the only mathematical
69    relationship I could think of to successfully
70    achieve this, and it is somewhat expensive in
71    comparison to the isUniqueInColVec and
72    isUniqueInRowVec implementations. *)
73 fun blockVec initVals culVec p i =
74     let val itVec =
75         Vector.fromList(
76             List.filter (fn k => k<=i)
77                 (List.concat(
78                     List.tabulate(3, fn k =>
79                         List.tabulate(3,fn l =>
80                             (((i div 9)*9)-9-(((i div 9) mod 3)-1)*9)
81 + (l mod 3)+(i mod 9)-(i mod 3)+9*(k mod 9)))))) ;
82         in
83             getVec initVals culVec p itVec
84         end ;
85
86 fun isUniqueInBlockVec initVals culVec =
87     fn p => fn i =>
88         let val block = blockVec initVals culVec p i in
89             isUniqueInVec block (Vector.length(block)-1)
```

```
90     end ;
```

C.6 Constant complexity function for cycling permutations

```
1  fun nextPerm inArr fArr i =
2    if i < Array.length(inArr) andalso (not (i = ~1))then
3      if i > Array.sub(fArr,i) then
4        (if (i mod 2) = 0 then
5          swapEle inArr 0 i
6          else swapEle inArr (Array.sub(fArr,i)) i;
7          Array.update(fArr,i, (Array.sub(fArr,i)+1)));
8        1)
9      else
10       (Array.update(fArr,i,0);
11       nextPerm inArr fArr (i+1))
12    else
13      ~1
```

C.7 Hamiltonian circuit generic search implementation

```
1  fun haCircuitAlt g : SearchProblem =
2    let fun testVertexPair (g : Graph ) (p : int->int) i j =
3          not ((p i = p j ));
4
5          fun testVertexPairsFrom g p i j =
6              i=j orelse (testVertexPair g p i j
7              andalso testVertexPairsFrom g p i (j+1)) ;
8
9          fun testVerticesFrom g p i =
10              i=(#size g) orelse (#isEdge g (p i) (p ((i+1) mod (#size g) ))
11              andalso testVertexPairsFrom g p i 0
12              andalso testVerticesFrom g p (i+1)) ;
13
14    in
15      dimensions = List.tabulate ((#size g), fn _ => (#size g)) ,
16      property    = fn p => testVerticesFrom g p 0
17    end ;
```

Appendix D

Additional data

D.1 Find all solutions for each search algorithm on a simple graph coloring problem

The naive, pruned, backup pruned and bespoke generic search algorithms were run on a simple graph to find all solutions. The left-most column gives the solutions that were returned by the bespoke and naive search algorithms, in the order that they were found (ordered from the top of the list). The right-most column gives the solutions that were returned by the pruned and backup pruned search algorithms.

[0, 1, 0, 2]	[1, 0, 1, 2]
[0, 1, 0, 3]	[1, 0, 1, 3]
[0, 1, 2, 3]	[1, 0, 2, 3]
[0, 1, 3, 2]	[1, 0, 3, 2]
[0, 2, 0, 1]	[2, 0, 1, 3]
[0, 2, 0, 3]	[2, 0, 2, 1]
[0, 2, 1, 3]	[2, 0, 2, 3]
[0, 2, 3, 1]	[2, 0, 3, 1]
[0, 3, 0, 1]	[3, 0, 1, 2]
[0, 3, 0, 2]	[3, 0, 2, 1]
[0, 3, 1, 2]	[3, 0, 3, 1]
[0, 3, 2, 1]	[3, 0, 3, 2]
[1, 0, 1, 2]	[0, 1, 0, 2]
[1, 0, 1, 3]	[0, 1, 0, 3]
[1, 0, 2, 3]	[0, 1, 2, 3]
[1, 0, 3, 2]	[0, 1, 3, 2]
[1, 2, 0, 3]	[2, 1, 0, 3]
[1, 2, 1, 0]	[2, 1, 2, 0]
[1, 2, 1, 3]	[2, 1, 2, 3]
[1, 2, 3, 0]	[2, 1, 3, 0]
[1, 3, 0, 2]	[3, 1, 0, 2]
[1, 3, 1, 0]	[3, 1, 2, 0]

[1, 3, 1, 2]	[3, 1, 3, 0]
[1, 3, 2, 0]	[3, 1, 3, 2]
[2, 0, 1, 3]	[0, 2, 0, 1]
[2, 0, 2, 1]	[0, 2, 0, 3]
[2, 0, 2, 3]	[0, 2, 1, 3]
[2, 0, 3, 1]	[0, 2, 3, 1]
[2, 1, 0, 3]	[1, 2, 0, 3]
[2, 1, 2, 0]	[1, 2, 1, 0]
[2, 1, 2, 3]	[1, 2, 1, 3]
[2, 1, 3, 0]	[1, 2, 3, 0]
[2, 3, 0, 1]	[3, 2, 0, 1]
[2, 3, 1, 0]	[3, 2, 1, 0]
[2, 3, 2, 0]	[3, 2, 3, 0]
[2, 3, 2, 1]	[3, 2, 3, 1]
[3, 0, 1, 2]	[0, 3, 0, 1]
[3, 0, 2, 1]	[0, 3, 0, 2]
[3, 0, 3, 1]	[0, 3, 1, 2]
[3, 0, 3, 2]	[0, 3, 2, 1]
[3, 1, 0, 2]	[1, 3, 0, 2]
[3, 1, 2, 0]	[1, 3, 1, 0]
[3, 1, 3, 0]	[1, 3, 1, 2]
[3, 1, 3, 2]	[1, 3, 2, 0]
[3, 2, 0, 1]	[2, 3, 0, 1]
[3, 2, 1, 0]	[2, 3, 1, 0]
[3, 2, 3, 0]	[2, 3, 2, 0]
[3, 2, 3, 1]	[2, 3, 2, 1]