An Investigation into Algorithms to solve the Snake-In-The-Box problem

Student Name: B.R. Rougier
Supervisor Name: D. Paulusma
Submitted as part of the degree of Bsc Computer Science to the
Board of Examiners in the Department of Computer Sciences, Durham University

Abstract—

Context

The Snake-In-The-Box problem (SIB) is a computationally complex path finding problem involving finding long induced paths or ”snakes” in m-dimensional hypercubes. It has been extensively studied for its uses involving grey codes, and more simply as a benchmark in testing path finding solutions.

Aims

The aim of this project is to understand the approaches previously taken to tackle this problem and then implement a series of algorithms to find solutions, so that experimentation and comparisons between approaches can be made, and algorithm variations can be explored.

Method

We achieved this by constructing a number of algorithms, starting with very basic approaches, for example a random search, before adding heuristic guidance, and eventually touching on far more complex algorithms such as the Nested Monte Carlo Search (NMCS) and Stocastic Beam search, which have previously proven effective at solving this problem. Experiments were then conducted on these algorithms to determine their most effective variants to solve the SIB problem. Several graph theory approaches and optimisations were also applied, specifically Kochut's constraint and snake priming, in order to test the improvement in performance they provided.

Results

Beam Search was found to be our most effective algorithm overall, and performed best using a combination of Legal node numbers and Narrowest Path playouts as a fitness function, with reverse tournament selection. Monte Carlo was also found to be effective and in rare (and specific) circumstances outperformed Beam Search. These graph theory optimisations greatly increased performance, although priming proved difficult to implement effectively.

Conclusions

In conclusion, these complex algorithms proved effective at solving the Snake-In-The-Box problem, showing the heuristic algorithm approach to be an appropriate solution, particularly for the lower dimensions. However more work in graph theory and algorithm efficiency is needed to improve performance for the higher dimensions.

Index Terms—Beam Search, Combinatorial optimization, Heuristic methods, Monte-Carlo Seach, Snake-In-The-Box, Heuristic methods

1 Introduction

The Snake-In-The-Box (SIB) problem was first introduced in 1958 by William Kautz [17], who created the problem whilst analysing error checking codes. Kautz found that a ”snake” could be viewed as a gray code, which in turn has significant uses in error-checking. SIB codes continue to be shown to have uses in electrical engineering [19], disk encoding [21], pattern recognition [26], rank modulation in flash memory [30] and even genetics, for modelling gene regulatory networks [31]. It follows that a longer gray code is more effective and hence the problem of finding the longest possible snake was formed.

Due to the difficultly of the problem, it has also found use simply as a benchmark in the effectiveness of path finding algorithms and approaches.

1.1 What is a Snake?

The SIB problem is a challenging combinatorial search problem. It consists of attempting to find the longest induced path through an m-dimensional hypercube. In other words, the path is a sequence of nodes in a hypercube, where each node in the sequence is connected by an edge, but each new node added to the sequence can not be connected by an edge to a previously visited node. Performance of an algorithm to solve this problem is measured using the longest length of the snakes they found, although this is dependent on the time taken. The complexity of this problem has been shown to be in NP-hard for bipartite graphs [14], however the complexity for hypercubes is not known.

1.2 Representation

The m-dimensional hypercube graph is represented by a set of binary numbered nodes from 0 to $2^n - 1$. Two nodes are then adjacent if they differ by only one bit and it is this property that allows the snake to be modelled as a gray code. For example, in a 4 dimensional hypercube the representation of the starting node 0 is ”0000”, which is then adjacent to 4 other nodes: (”0001”,”0010”,”0100”,”1000”).

There are also multiple methods of representation for a snake. The most simple of these is the node representation, where a snake is a list of integers representing each node in the path. However the transition sequence representation, which records the snake in a list of integers showing which bit changes in from one node to the next, tends to be favoured because it assists in maintaining node adjacency in the crossover of genetic algorithms (which have been commonly applied to the SIB problem) and more clearly illustrates the patterns present within snakes.

Figure 1 shows an optimal valid snake in the 3 dimension hypercube. This snake begins at node 0 (000) and has a transition sequence of 0,1,2,0.

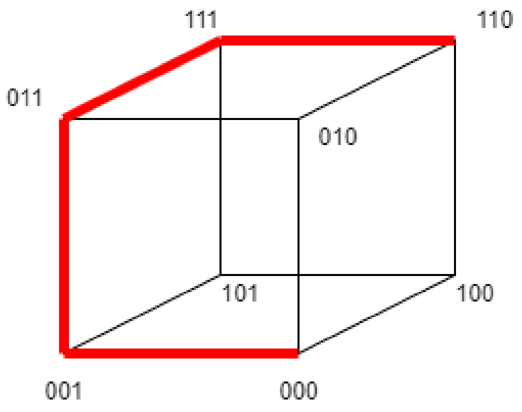

Fig. 1. An optimal snake in dimension 3

Figure 2, on the other hand, depicts an illegal/invalid snake. It copies the path seen in Figure 1, but adds one additional transition. It is illegal because the node travelled to (100) is a neighbour of a node already present in the path (000).

1.3 The Challenge

Whilst looking at small dimensions of hypercube, the SIB problem appears to be fairly simple. However it is afflicted with combinatorial explosion, which describes problems which have a massive increase in complexity as bounds are increased. In the case of SIB, the search space of the hypercube increased exponentially as the dimension is increased. As a consequence, for the higher dimensions, basic exhaustive searches are not possible and graph theory techniques and heuristic algorithms must be used. This in turn raises another complication, as unlike in similar path finding and search problems, like the Travelling Salesman Problem, there are no weights/distances for the edges connecting nodes, making collecting heuristic data a far less obvious task.

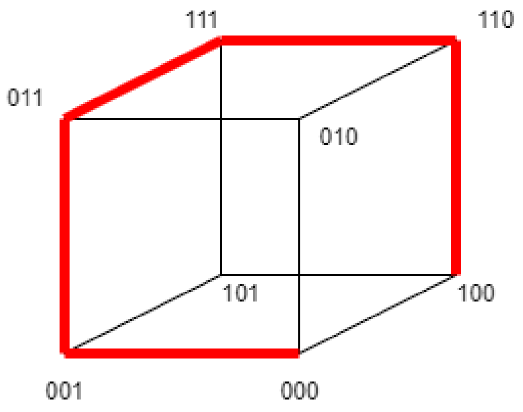

Fig. 2. An illegal/invalid snake in dimension 3

1.4 Coil-In-The-Box

Coil-In-The-Box (CIB) is a closely related problem to the SIB. It searches for the longest induced cycle through an m-dimensional hypercube and so differs in that all snakes must finish at the same node they started on, forming a coil, hence the name. Obviously, for traditional snakes this would not be possible, as an ”illegal” node must be travelled to in order to return to the starting node. Therefore an exception is made for the CIB problem.

It is also an extensively studied problem, and most algorithms implemented for SIB can be easily modified to search for a coil.

Figure 3 displays an optimal coil for the 3 dimensional hypercube. Notice the coil contains the optimal snake seen in Figure 1.

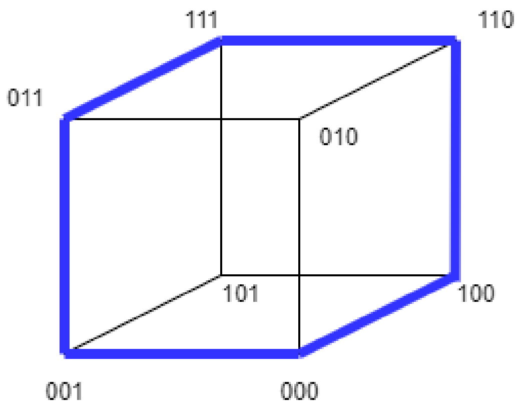

Fig. 3. An optimal coil in dimension 3

1.5 Objectives and Accomplishments

The main objective of this project is to investigate and compare the performance of different algorithms and approaches to the Snake-In-The-Box problem. In order to achieve this, objectives were split into three categories: minimum, intermediate, and advanced.

The minimum objectives were firstly to understand the problem, and the previous work that has been done to solve it. Once this had been accomplished we aimed to begin coding solutions, starting with a simple checker to confirm if a snake was valid, and then venturing into simple algorithms to generate snakes. This included Random Walk, which randomly picked available nodes, Narrowest Path Walk, a simple greedy algorithm, and basic exhaustive solutions. The purpose of this was to build understanding of the problem and form a benchmark for performance through testing and comparison. All of these aims were accomplished.

Intermediate objectives became more complicated, we picked two more complicated algorithms, Beam Search and Monte-Carlo Search, and implemented them for the SIB problem. These were chosen as they had previously been shown to be effective for this problem. Variations and experimentations for these algorithms would then be necessary to allow for comparisons to determine the most effective solution and approach. On top of this, certain graph theory optimisations to further test performance impact should be added and the simple algorithms improved. Extending the algorithms to look at the related problem, Coil-In-The-Box, was also an intermediate objective. Again, these were all achieved.

Finally the advanced objectives were to create exceptionally well performing solutions, nearing or even improving on the current records. This would likely need to be done using Hamiliton (Durham University's high performance computing service). Our completion of these objectives was not as clear cut. Our solutions did not improve any current records, but did equal and come close in some scenarios. Our algorithms were also not run for anywhere near the time record breaking runs have been in the past and were coded in python, making like for like comparisons difficult.

2 Related Work

Since it's introduction in 1950, the Snake-In-The-Box problem has been extensively researched. The approaches taken to solving the problem can be broadly placed into two categories, although there is often huge overlap between them. These are heuristic based approaches and graph theory approaches and optimisations.

2.1 Heuristic Approaches

A large number of heuristic path finding algorithms and approaches have been implemented for the SIB problem, with varying success.

2.1.1 Genetic Approaches

Genetic based algorithms are one of the most common approaches to tackling the Snake-In-The-Box problem. The most simple of these is the basic genetic algorithm. First proposed by J.H Holland in 1975 [16], the genetic algorithm is a stochastic and heuristic algorithm based on Charles Darwin's theory of evolution. It a used in a large variety of optimisation and search problems. The algorithm follows evolutionary principles using a fitness based selection for members of a population. In a basic genetic algorithm, there are three operators: the selection of the "fittest" members of the population for reproduction, the crossover, and the mutation. Once an initial population has been established, the 'fit' members of said population are used as "parents" to create "child" paths. These are paths created using the crossover of two parents. A small number of random mutations might also be added to children to further increase variety in the children.

Implementations for the SIB problem have been shown to be very effective. Potter et al. were the first to find the longest possible snake for dimension 7 (50) using the genetic algorithm [25] and since then many have improved on their results for greater dimensions, including Bitterman in 2004 [6], Carlson in 2010 [7], and Rashmi et al. in 2019 [1].

Another variation of genetic algorithms is the Hill Climber Algorithm, a common path finding and optimisation algorithm that can also be applied to the SIB problem. In 2005, Casella and Potter [8] successfully used a Population Based Stochastic Hill Climber (PBSHC) to find new lowest bounds for snakes (at the time) for dimensions 9-12. The algorithm itself is very similar to the basic genetic algorithm, except with a few key differences. Firstly, the selected population "reproduce" asexually, without need for a crossover function. There is also a "growth operator" which randomly picks the next node of the hypercube for each snake's path to "climb" to.

Casella notes that the lack of a crossover function is often considered a disadvantage of the PBSHC against the genetic algorithm, however they did not find it to be the case for the SIB problem and proved this by outperforming all previous attempts to solve the SIB for the dimensions 8 and 9.

2.1.2 Beam Searches

The term Beam Search was first coined by Reddy in 1977 [11]. The algorithm is effectively an optimisation of Breadth-first search, with heuristically guided pruning to reduce the size of the search space at each level. It was first applied to the Snake-In-The-Box problem in 2007 by Tuohy et al. [28], although in this case it was not referred to as a Beam Search but an "Evolved Pruning Model". Tuohy's implementation set record lower bounds for dimension 8 and 9.

More recently, Meyerson et al. adapted the Beam Search for the SIB problem in their 2014 paper [22] and then later expanded on this work to find new lowest bounds for dimensions 11-13 [23]. In 2016 Allison and Paulusma [3] used an adaptation of this stochastic beam search method to find record lowest bounds for the snakes in dimensions 11- 13. These records still stand, proving that Stochastic beam search is a hugely effective method for solving the SIB problem.

2.1.3 Monte Carlo Searches

The use of a Monte Carlo Search for the Snake-In-The-Box problem is a very unique approach in comparison with those previously mentioned. The algorithm is more commonly used as a heuristic decision making algorithm, most famously focusing on board games [13]. However, a variant of Monte Carlo Search has been adapted to the SIB problem by Kinny in 2012 [18]. His implementation of Nested Monte Carlo Search proved very effective, setting record lowest bounds for dimensions 10, 11, and 12, although for 11 and 12 these have since been beaten. We elaborate more into the technical implementation of the algorithm in our methodology.

2.1.4 *Other heuristic approaches*

Whilst the previously mentioned algorithms are the most effective and common past implications, approaches with most well known heuristic algorithms have been attempted.

Hardas implemented an Ant Colony approach to the problem in 2005 [15]. The Ant Colony search is a path finding search algorithm first proposed by Dorigo in 1992 [10], inspired by the way ant colonies use pheromones to forage in the wild. The algorithm has proven been very successful for comparable problems like the Travelling Salesman Problem [12], and produced good, but not record breaking results for the SIB problem.

Other nature inspired searches, such as simulated annealing search, an algorithm inspired by annealing in metallurgy, and particle swarm optimisation, modelled on the motion of flocks of birds, were implemented for SIB in 2009 by Atluri [4], who was able to find near optimal 8 dimension snakes with these algorithms. However, in comparison with other approaches released at a similar time, such as Carlson's genetic approach [7], results were underwhelming, which suggests these algorithms might not be the best suited to the problem.

2.2 Graph Theory Approaches and Optimisations

2.2.1 *Kochut's Constraint*

Kochut's Constraint is a constraint on the growth of Snake paths designed to eliminate isomorphic duplication due to the inherent symmetry of the hypercube, and hence significantly reduce the size of the search space. A snake following this method can be described as being represented "canonically". The method was introduced by Kochut in 1996 [20], where this constraint was used in conjunction with an exhaustive search to prove the maximum length snake and coil for dimension 7. Since then, every record breaking SIB implementation has used the constraint.

Figure 4 depicts the constraint in action. Notice the snake displayed is isometrically identical to the snake displayed in Figure 1, although the nodes of the snake, and therefore transition sequence, are different. This duplication is not possible following the the rules of the constraint.

2.2.2 *Graph Based Records*

In 1991, Abbott and Katchalski [2] were able to use hypercube symmetry properties, combined with extension of lower dimension snakes to construct record setting lower bounds, some many of which are still standing for dimensions ≥ 14. This 30 year unbeaten stand for Abbott and Katchalski's results is extremely impressive given the huge advancements in computing power available since 1991 and

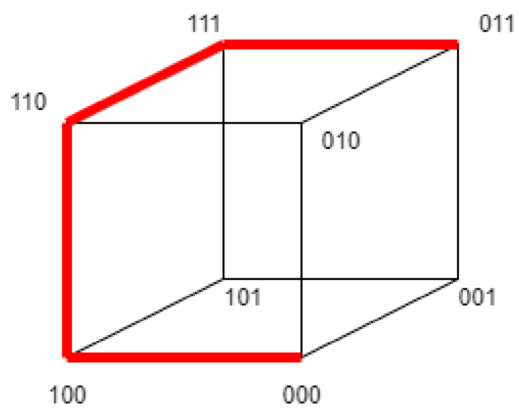

Fig. 4. An example of isometric duplication

considering the records for dimensions $8 - 13$ have been consistently improved throughout.

It exposes a possible weakness or limit in the performance of heuristic approaches for the high dimensions, given their failure to outperform such an old graph theory based approach with modern technology.

Using a similar strategy of symmetry in order to optimise an exhaustive search, Ostergard and Pettersson [24] used canonical augmentation to prove the maximum possible snake and coil for dimension 8.

2.3 Current known bounds

Due to the nature of the problem, finding the longest possible path becomes exponentially more computationally difficult as the number of dimensions increases. Currently there only exists a definite known longest path for the first m dimensions, where $m \leq 8$ and it could be stated that the SIB problem has been solved for these dimensions. For $m \geq 9$ we only have the current lowest known bounds of the longest possible snake.

- For $m \leq 6$, Davies used an exhaustive search to find longest possible snakes
- For $m = 7$, Potter, Robinson, Miller, Kochut, and Redys [25] used a genetic algorithm to find the longest snake . This was later confirmed using an exhaustive search.
- For $m = 8$, Östergård and Pettersson [24] used the exhaustive search method of canonical augmentation to find the longest possible path
- For $m = 9$, Wynn [29] set the current lowest known snake bound of 190 using sequence permutation in 2012.
- For $m = 10$, Kinny [18] holds the record bound of 370, using Nested Monte Carlo Search in 2013.
- For $11 \leq m \leq 13$, Allison and Paulusma [3] used Stochastic beam search to generate best lower bounds.

- For $14 \leq m \leq 20$, Abbott and Katchalski's [2] implementation using hypercube symmetries are still unbeaten, 30 years later.

Table 1 displays there records for the first 15 dimensions. A * indicates the snake has been proven to be optimal.

TABLE 1
Current best lower bounds of snakes

Dimension	Length of record snake
1	1*
2	2*
3	4*
4	7*
5	13*
6	26*
7	50*
8	98*
9	190
10	370
11	712
12	1373
13	2687
14	4932
15	9866

3 Methodology

Our solutions designed to solve the Snake-In-The-Box problem are outlined here. The approach taken was to begin with more simple approaches, before implementing our more complex algorithms. This approach was taken for a number of reasons. Firstly, beginning with simple algorithms helped us better learn the intricacies of the problem early and allowed us to conduct quick testing and optimisations. Secondly, the more simple methods serve as a benchmark to compare other algorithms against. The final reason was, as we will elaborate more on later, some of our more basic approaches are used as aspects of the more demanding algorithms.

3.1 Implementation tools

We decided to use python coding language (version 3.10) to generate our snakes. This decision was made as the main objective of the project was to understand and compare the performance of algorithms to solve the SIB problem. Therefore it was not as essential to use faster coding languages, like C++ or java, and therefore we chose python for ease of use, which allowed us to conduct more experimentation.

However, despite this trade off, reasonable algorithm speed and longer path lengths were still desirable. Therefore instead of using the default python interpreter, CPython, we decided to use PyPy [27]. PyPy is a replacement interpreter that uses just-in-time compilation which we found resulted in an approximate 3x speed increase for our use cases in comparison to CPython.

Numpy is often viewed as a faster and superior alternative to python's inbuilt list functionality. However, surprisingly, we did not find that to be the case for our implementation. Whilst it is true that numpy is significantly faster for mathematical operations, we did not use these in our approaches and found numpy to be slower for for array augmentation and indexing than traditional lists [5], which were the two functions we required for our implementations.

Our own testing supported this, and as a consequence numpy was not used.

3.2 Graph theory optimisations

3.2.1 Kochut's Constraint Implementation

Kochut's constraint [20] for canonical representation aims to eliminate isomorphic duplication and massively reduce the search space of the SIB problem. It is achieved by insuring that a snake does not travel to a dimension until the previous dimension has been explored. In transition sequence terms, this means that the transition number can not be more than 1 higher than the highest number that has already occurred. For example transition sequence "024", which represents the path "00000, 00100, "10100", does NOT follow Kochut's constraint. In fact, using canonical representation, in dimensions 4 and greater, the transition sequence will always start 0120 or 0123. All of our algorithms were tested with and without Kochut's constraint in order to determine the effect on the performance of each algorithm.

3.2.2 Priming

Snake-In-The-Box priming refers to choosing snakes from lower dimensions as a starting point for higher dimension searches. The aim of priming is to dramatically reduce the search space of algorithms using a fixed starting point. Long snakes from previous dimensions also tend to perform well as primers, however this is not always the case. It has been proven that none of the optimal length 50 7-dimension snakes can be extended into optimal 8-dimension snakes [18]. Therefore it is necessary for a selection of near optimal snakes to be taken. Priming can be performed using a single snake, or multiple, dependent on the algorithm. Beam Search, for example, can use a large population of primers, whereas Nested Monte Carlo Search requires just a single snake.

This raises a problem: In situations where only one snake can be used as a primer, how does one ensure that the correct snake is chosen?

We found this to be a significant weakness of algorithms with this requirement, as this is an extremely difficult problem to solve. One solution we found was to simply run the algorithm many times, and each time use a different snake as a primer. However, this approach has obvious shortcomings. If the population of near-optimal snakes is high, and the algorithm used has a long run time, this approach can take an unacceptably long time.

Another approach was to use an algorithm capable of accepting a whole population of primers, in order to narrow down the population. This approach was very dependant on the performance of the algorithm used.

The final approach was to rank the population, using averaged results from biased walk playouts, which will be further discussed later, and then discard the lowest ranking paths. This was the approach we eventually settled on, as it was by far the most efficient, and showed effective performance.

Whilst priming is not an approach as widely used as Kochut's constraint, it has been shown to result in faster run times and longer snakes found [18]. We tested our complex algorithms with and without priming, to measure the performance difference. We also used cherry picked primers in testing. Cherry picked primers were taken from previously found successful and recording breaking snakes, and so the results of this experiment must be taken with a pinch of salt.

3.3 Simple Search algorithms

3.3.1 Random Walk

The simplest of our algorithms is the Random Walk algorithm. This is a purely stochastic algorithm which generates a snake by starting at a set node (the "head" of the snake) and generating a list of all available next nodes to add to the snake. The next node is then chosen at random from this list to become part of the snake and the new head, and this process is then repeated until the snake reaches a *deadend*, which is defined as a node where none of the neighbouring nodes are available, at which point it stops.

Algorithm 1 Random Walk

```
head = starting node
snake = head
while $head \neq deadend$ do
  Find all available next nodes from head
  if number of availble next nodes = 0 then
    head = deadend
  else
    head = random selection from available nodes
    Append head onto snake
  end if
end while
return snake
```

3.3.2 Narrowest Path Walk

The Narrowest Path Walk (NPW) [6] is largely similar to the random walk, with one key difference. It introduces a heuristic which is the basis on which almost all SIB heuristic algorithms rely on. Instead of randomly picking from the available nodes, the Narrowest Path Walk performs a "lookahead" which finds the available node which itself has the smallest number of available nodes itself (also known as the nodes branching factor), as long as this is greater than 0. If the branching factor was 0 this would signify a deadend node and hence would not be useful to travel to. Whilst this might seem counter intuitive at first, due to the property of the SIB problem where all neighbours of a chosen node become unavailable once it is a part of the snake, taking the option with the smallest number of available nodes minimises the number of nodes that become "illegal".

Algorithm 2 Narrowest Path Walk

```
head = starting node
snake = head
while $head \neq deadend$ do
  Find all available next nodes from head
  if number of available next nodes = 0 then
    head = deadend
  else
    for Each available node do
      Find the branching factor
    end for
    head = node with smallest branching factor ($> 0$)
    Append head onto snake
  end if
end while
return snake
```

This basic NPW algorithm involves no randomness, and therefore always finds the same snake for each dimension.

3.3.3 Biased Walk

Through testing (see results section for more detail) I found that the snakes generated by NPW were always longer than the mean of the random walk results, however in some cases the random walk was able to generate individual snakes longer than that of NPW (given enough runs). It therefore became necessary to combine the strengths of both of these algorithms: the randomness of random walk and the heuristic guidance of narrowest path walk. Therefore we implemented Bias Walk [18], which at each step has a n probability of either picking a node at random or selecting the node with the least available nodes. Bias Search with n=0 is effectively the same as NPW and with n=1 is the same as Random Walk. Extensive testing was then conducted to find the best value of n for the different dimensions.

Algorithm 3 Biased Walk

```
head = starting node
snake = head
while $head \neq deadend$ do
  Find all available next nodes from head
  if number of availble next nodes = 0 then
    head = deadend
  else
    Generate a random number p between 1 and 0
    if p $\leq$ n then
      for Each available node do
        Find the number of available nodes
      end for
      head = node with least available nodes ($> 0$)
      Append head onto snake
    else
      head = random selection from available nodes
      Append head onto snake
    end if
  end if
end while
return snake
```

3.3.4 *Depth-first search and Breadth-first Search*

We decided to implement two simple exhaustive searches for the SIB problem. This was done in order to determine and demonstrate at which dimension basic exhaustive searches become obsolete due to the combinatorial explosion of SIB. It also serves as a method for analysing the effectiveness of Kochut's Constraint in reducing the search space by preventing isomorphic duplication and the effects of priming.

Depth-first and Breadth-first Search are both exhaustive tree search algorithms. Depth-first search (DFS) starts at a root node and explores as far as possible along each branch, until a deadend is hit, at which point the algorithm backtracks. Breadth-first search (BFS), on the other hand, starts at a root node and at each level explores every node before moving on to the next level. It is fairly clear that DFS should be better suited to the SIB problem, as DFS can arrive at optimal solutions before the whole search space has been explored, which BFS can not, and in the case the algorithms are terminated before the entire search space is explored, DPS is likely to return the longer solution due to its priority of depth.

Straightforward adaptations of BFS and DFS were implemented for the SIB problem, both with Kochut's constraint and without, and with primed snakes and without to compare performance impacts of these criteria.

3.4 Beam Search

Beam search is a greedy heuristic search method based on the best-first search algorithm. Our implementations are based on that of Meyerson et al. [22], who used the algorithm to set a number of records for the SIB problem in 2014 and 2015. While the algorithm is not ″complete″ and therefore cannot guarantee optimal snakes, it has previously been shown to produce excellent results for problems with rapidly expanding search spaces.

The beam search algorithm functions similarly to the breadth-first search previously mentioned, however it introduces measures to reduce the search space. The algorithm progresses through levels in a search tree, and at each level adds every possible path to memory. These paths are then evaluated and the ″worst″ paths are pruned. The number of paths pruned is dependent on the *beam width* of the algorithm: This is the maximum number of paths that remain at each level.

Figure 5 depicts the first 3 levels of the Beam Search algorithm with a beam width of 2, in a problem with a branching factor of 4. The red nodes depict those that have been pruned and the green those that are chosen to continue to the next level of the algorithm.

3.4.1 *Implementation for SIB*

The implementation of a Beam Search for the SIB problem can be split into 4 distinct sections.

Initialisation: Like with all of our previous algorithms, beam search is initialised from the 0 node, so that it can be kept in Canonical Form.

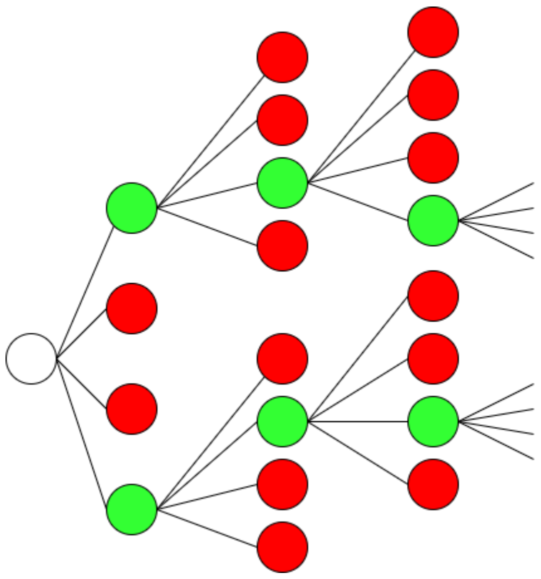
Fig. 5. Beam Search Algorithm with a beam width of 2

Path Generation: The algorithm must generate every possible snake for the next level from the population of snakes not pruned in the previous level. This is done in the same way as in Breadth-first search: Every node adjacent to the current head of the snake is check to ensure it is not an ″illegal″ node. If it is legal, this new node is added to the snake and the snake is added to the new population. This is done for every snake in the previous population and creates a new population of snakes exactly one node longer than the previous level.

Evaluation: This new population of snakes must be evaluated. However, the fitness function used to evaluate it can be implemented with a number of different heuristics and methods, which is one of our key areas of experimentation in the algorithm.

Selection and pruning: The number of snakes must now be pruned to the size of the beam width. The most obvious way to do this is to simply rank every snake by their fitness function score and prune those that scored the worst. However this is not the only possible approach, and is another area of experimentation. ********

3.4.2 *Fitness Functions*

Implementations were constructed for Beam Search using 4 different methods of fitness function evaluation:

1. Legal nodes: The first algorithm created used the number of legal nodes in the hyper-cube to evaluate the snakes. This fitness function was the basis of the record breaking algorithm by Meyerson et al. [22], albeit with a few alterations. The heuristic calculates the number of individual ″skin nodes″ of a snake, which are nodes adjacent to nodes already in the path, and therefore illegal. The theory behind this method is that the smaller the number of individual skin nodes (caused by multiple nodes in the snake sharing the same skin nodes), the greater the number of legal nodes left in the hypercube. More legal

Algorithm 4 Beam Search

```
head = starting node
snake = head
population = [snake]
while population size $\neq 0$ do
    for Each snake in population do
        Find all available next nodes from head
        if number of availble next nodes = 0 then
            Remove snake from population
        else
            for Each available node do
                Evaluate fitness
            end for
        end if
    end for
    Perform selection
    Prune lowest fitness snakes in selection
end while
return Longest found snake
```

nodes in the hypercube translates into more space for the snake to grow into and hence longer snakes. However this approach is not without its faults. Whilst available nodes is a good metric, it fails to account for whether an available node is actually reachable by the snake - the node might be surrounded by illegal nodes and is therefore useless. Also, minimising the number of skin nodes is a high risk, high reward approach. Whilst paths that contain low numbers of individual skin nodes can produce excellent snakes, these paths are also more likely to lead to deadends as a consequence of the tight winding they causes.

2. Playouts: Our next fitness ranking uses snake lengths generated by playouts to rank snakes. This involves taking the snakes currently in the population and using the simple search algorithms previously described to extend them. To perform the payout I tested Random Walk, Narrowest Path Walk and Biased Walk (with varying probabilities of $0 < n < 1$). I found that that Narrrowest Path Walk performed the best in this scenario and was therefore chosen. Whilst the randomness of Biased Search can sometimes generate longer, less obvious paths, in this application I found it was also much more likely to playout poorly on strong paths, reducing the overall effectiveness of the beam search pruning.

Using playouts as a fitness function also had its flaws. NPW, whilst better than randomness, rarely produces the best possible path from a given node, and therefore could often score strong paths poorly.

3. A Combination of Legal Nodes and Playouts: In an attempt to mitigate the flaws of the two previous evaluation methods, a combination of the two was implemented. First, a shortlist of paths to prune was created using the available nodes heuristic, and this shortlist was then ranked using the playout method.

Whilst using a double evaluation obviously results in greater computational expense, the hope is that the performance increase of this algorithm would make this worth while and outperform the time increase of simply increasing beam size to achieve better snakes.

4. Random Picking: An evaluation using randomised picking to rank the paths was implemented as a benchmark for comparison against the previously mentioned methods.

3.4.3 Selection Methods

2 methods of path selection were implemented and tested for the Beam search SIB algorithm:

1. Total Path Selection: The most obvious method is Total Path Selection (TPS). This is where every path in the population is selected and ranked against each other, and only the best ranked n are not pruned, where n is the beam width. This method insures, at every level, that only the heuristic best paths are continued, with no randomness.

2. Reverse Tournament Selection: Reverse Tournament Selection (RTS) was the selection method favoured by Meyerson et al. [22] in their beam search implementation. Paths are randomly selected from the population for the tournament, where they are ranked and the worst performing are discarded. This is then repeated until the number of paths left in the population is less than the beam width. There are two settable parameters in this selection method: the size of the tournament (number of paths), and the amount of discarded paths per tournament. The ratio between these two numbers controls the amount of randomness introduced into the selection progress. I found that a tournament size of 50, with 5 discarded produced the best results, a ratio of 10:1. If this ratio was any lower, too much randomness resulted in discarded good paths; if it was any higher the selection effectively performed the same as Total Path Selection and the computation expense vastly increased.

3.5 Nested Monte Carlo Search

Nested Monte-Carlo Search (NMCS) is a recursive tree search algorithm. It is a variant of Monte-Carlo Search, a heuristic search algorithm commonly used in problems with decision processes. The algorithm might be best known for its use in Alpha Go, Google's record breaking Go program [13]. The nested variant which we have implemented was created by Cazenave in 2009 [9] and specialises in problems with limited heuristics. It has previously broken records in single player games. Our implementation of NMCS is inspired by Kinny's record breaking approach in 2012 [18], although Kinny did not provide significant details of his approach, and as a consequence much of our implementation might be different.

NMCS functions by selecting a node, and recursively creating a k level search tree. Playouts from the k level then determine which next step to take from the selected node. In contrast to Beam Search, NMCS is an "anytime algorithm", meaning it can be stopped at any point, still returning a valid snake, and will run indefinitely, or until it has searched the entire search space.

Figure 6 depicts a general example for a 2-level NMCS. The selected node is 1, and then a two level sub-tree is constructed. Playouts are conducted from the leaves of this subtree (nodes 4, 5, 6 ,7). The playout from node 6 scores the best, and as a consequence node 3 is added and selected for use in the next iteration.

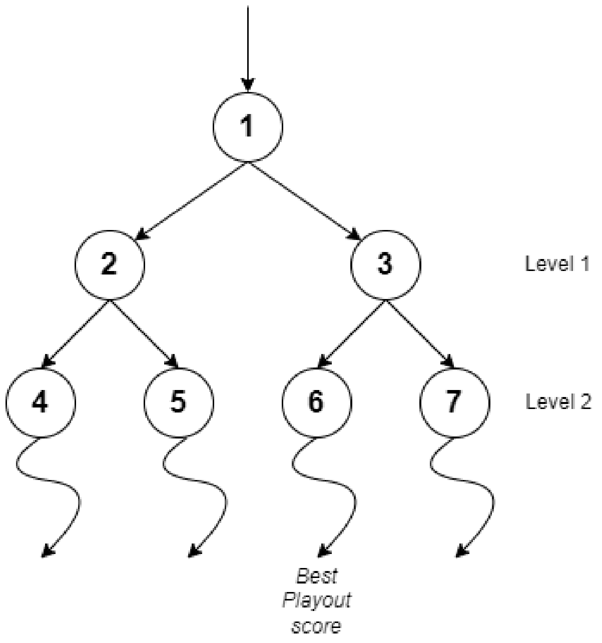

Fig. 6. A general 2-level Nested Monte-Carlo Search example

3.5.1 The need for the nested approach

As previously mentioned, NMCS is a variant specialised to problems with limited heuristics, which begs the question: Given the existence of the number of legal nodes node heuristic, why is this variant preferred for SIB? The answer comes down to the relationship between the legal nodes heuristic and the lengths of the snakes. The heuristic works well at judging between snakes when they are all the same lengths, but has been shown to be very hard to standardise across different lengths snakes [28] due to not apply uniformly as depth increases. Another reason is the high risk of the heuristic, which makes deadends more likely. This risk can be mitigated in Beam Search, by keeping a large population, but not for Monte-Carlo Search. Therefore, NMCS is the obvious choice for SIB.

3.5.2 Implementation for SIB

The implementation of a NMCS for the SIB problem can be split into 4 distinct sections:

Selection: If the algorithm is in the first iteration the zero node is selected, in order to maintain canonical form. Otherwise the legal leaf node leading to the current best found score is selected.

Create Nested Tree: A k-level subtree of the search space, starting from the selected leaf node is made. First all legal nodes of the selected nodes are found (creating a 1 level search tree) and then recursively all legal nodes of these nodes are found until K levels are reached. This effectively results in a population of snakes, which differ only by the last k nodes.

Playouts: NMCS then uses playouts, from the snakes in the population, in order to score them based on the length of the snake the playout produces. The all-time best path found in these playouts is memorised, and used to select the next node. If none of the playouts from a population produce a longer snake than the all-time best path found, the all-time best is still used select the next node. This insures a successful snake is not wasted. To perform the playouts I used my Biased Walk with $0 \leq n \leq 1$.

Back-propagation and deadends: The scores for all possible paths tested are memorised. When the algorithms hits a deadend (when the selected path can no longer be extended) the path backtracks to the next most promising path that is not currently in memory. This continues indefinitely or until the entire search space has been searched.

3.5.3 Experimentation and testing

NMCS does not have many tunable parameters, only the selected level of the search and value of n in the Bias Search playouts, and so this is where the majority of our testing with the algorithm took place.

One observation we made was that, when the algorithm hit a deadend and backtracked to the next most promising node, nodes that were at a similar length to the current best found snake were very unlikely to find improvements, whereas nodes at approximately 2/3rds of the length on a path seemed to be much more successful. Therefore, our algorithm was limited so that only nodes before 2/3rds of the current record were saved into memory. This adjustment led to, on average, run-times decreasing for lengths of snakes founds.

3.6 Coil-In-The-Box

SIBs related problem, Coil-In-The-Box (CIB), at first seems a much more complicated task. It requires the the snake found have a finish point, the 0 node, which is also where the snakes begins. However the implementation is largely the same, except with one large alteration. In order to return to the starting node the coil must reach one of the neighbours to the starting node. These are known as ”target nodes” For the 5-dimension cube these nodes are ”00001”,”00010”,”00100”,”01000”,”10000”.

To modify our current algorithms for Coil-In-The-Box, anytime a node is added onto a snake a check is performed to determine the number of ”1” bits in the node. If there are only 2 ”1” bits it is a neighbour of two of the target nodes. A check is then conducted to determine if the connected target nodes have already been visited, and if not a coil is found.

As this is alteration is simply an extra check placed on each stage of a Snake-In-The-Box algorithm, the performance of algorithms for CIB is directly linked to SIB performance and should not be interesting for further investigation. Despite this, versions of both Beam Search and Monte Carlo for CIB were implemented and testing confirmed this theory.

4 Results

4.1 Testing Hardware

All test were conducted on a single core of an i5 7600k CPU and 16GB of RAM. We were careful to ensure a minimal number of background processes were running in order to maintain consistency between simulations.

4.2 Random Walk and Narrowest Path Walk

In testing, Random Walk was run 100,000 times for each dimension, allowing us to find both the maximum length snake the algorithm is likely to achieve and the mean length of all the runs. This was not necessary for Narrowest Path Walk, as the lack of randomness meant the outcome was exactly the same every run. The performance is measured using the length of the transition sequences found.

TABLE 2
Performance of Random Walk Algorithm

Dimension	RW (best)	RW (mean)	Time taken(s)
5	13	10.25	2.23
6	25	17.20	4.51
7	45	29.29	9.21
8	80	46.46	19.51
9	131	77.27	41.63
10	210	130.08	85.53

TABLE 3
Performance of Narrowest Path Walk Algorithm

Dimension	NPW
5	11
6	22
7	40
8	61
9	122
10	222

As NPW was only run once, the time taken for these results was very small ($< 0.1s$). However it is worth noting that the computational expense for NPW is significantly higher than RW. For example, when NPW is run for 100,000 iterations for the dimension 10 hypercube, the time taken is 299.06s, much longer than the 85.53s seen for RW (an approx. 3.5x increase).

These results show that the performance of NPW is substantially better than the mean of RW, but falls behind the best snake for all dimensions tested with the exception of the largest (10).

4.3 Biased Walk

Testing and experimentation for Biased Walk was centered around the value of n. The higher the value of n, the greater chance of picking a node randomly. N was tested at a series of values $0 < n > 1$, again with 100,000 iterations at each value, in order to find the best performing point. The results can be seen in table 4, in the form: best length found/mean length found.

TABLE 4
Performance of Biased Walk Algorithm for varying n values

Dimension	n = 0.05	n = 0.1	n = 0.3
5	13 / 10.91	13 / 10.83	13 / 10.56
6	25 / 21.18	25 / 20.50	26 / 18.72
7	46 / 37.16	46 / 36.18	47 / 31.58
8	85 / 59.38	85 / 57.92	84 / 53.79
9	151 / 104.70	149 / 99.52	148 / 91.88
10	272 / 174.34	273 / 167.98	259 / 156.93
Dimension	**n = 0.5**	**n = 0.7**	**n = 0.9**
5	13 / 10.44	13 / 10.36	13 / 10.29
6	26 / 17.92	26 / 18.57	26 / 17.33
7	47 / 30.31	47 / 29.54	46 / 28.73
8	82 / 51.64	80 / 49.76	77 / 47.55
9	141 / 87.83	137 / 83.77	128 / 79.48
10	242 / 149.51	228 / 141.80	221 / 134.16

The cells coloured in green represent the best found path for that dimension with the Biased Walk. This shows an interesting pattern; for dimensions 6 and 7 the smaller values of n found worse snakes than the larger, but for the higher dimensions the smallest value of n, 0.05, performed the best. The smaller the value of n, the better the mean length throughout the dimensions.

Another interesting find was that, for n = 0.9, Biased Search's best snakes found were actually worst than Random Walk for dimensions 8 and 9. This indicates that a small chance of heuristic guidance can actually be detrimental to performance in some cases.

The use of Kochut's constraint was not found to provide any meaningful changes in the performance of Biased Walk (or RW and NPW). This is likely due to the algorithms not being reliant on search space size.

4.4 Breadth-first Search and Depth-first Search

BFS ans DFS were tested in order to investigate their limitations as basic exhaustive algorithms to solve the SIB problem, the effects of Kochut's constraint on their performance, and the effects of priming. The algorithms were capped to a 500s runtime in testing due to their exhaustive natures, with red indicating non-optimal solutions.

First, BFS and DFS were run in their most basic form: without Kochut's constraint or any priming. These results are shown in Table 5.

TABLE 5
Performance of Basic BFS and DFS algorithms

Dimension	BFS	Time(s)	DFS	Time(s)
5	13	0.245	13	0.02
6	15	436.6	26	8.03
7	11	149.5	47	111.81

Both algorithms explored the entire dimension 5 search space extremely quickly. However this implementation of

BFS was only able to reach a length of 15 in dimension 6 in 500s, and only reached a length of 11 in dimension 7. It is also worth noting that the time between lengths for the BFS increases exponentially for most of the algorithm due to the > 1 branching factor, so this time of length 15 in 500s does not scale linearly when increasing length. BFS also introduced issues with RAM usage, as the algorithm required hundreds of millions of paths to be stored in memory.

DFS on the other hand performed significantly better, finding the optimal solution for dimension 6 in 8 seconds, and achieving a much better 47 length snake in dimension 7.

The results once Kochut's Constraint is added, in Table 6, show a massive improvement for BFS and demonstrate just how significantly canonical form reduces the search space. DFS's performance is interestingly far less effected, possibly because it is less search space size dependent.

TABLE 6
Performance of Canonical BFS and DFS algorithms

Dimension	BFS	Time(s)	DFS	Time(s)
5	13	0.04	13	0.03
6	26	4.58	26	5.23
7	17	96.41	47	106.2

Using near optimal dimension 6 snakes as primers, DFS (with Kochut's Constraint) was able to find an optimal 50 length same for dimension 7 in under 3 minutes. The big challenge for this was the trial and error involved in sorting through the primers to find one capable of an optimal dimension 7 path, as DFS can only be primed with a single snake, rather than a entire population, as with BFS.

While priming did improve the performance of BFS, it still suffered the significant issues as before with the exponentially expanding search space, and was therefore still unable to produce competitive snakes for any dimension greater than 6.

4.5 Beam Search

To determine the best performing variant of Beam Search simulations were first run on four implementations with the different fitness function previously mentioned. These tests were all run with a beam width of 1000 and the Reverse Tournament selection method. The fairly low beam width allows for more efficient testing, whilst remaining high enough to achieve reasonable results. Kochut's constraint was also implemented.

These results illustrate that using a combination of the Legal Nodes and Playouts fitness functions produces the best results. However, this approach does greatly increase the running time of the algorithm in comparison to the use of just the legal nodes fitness function. Further testing was then conducted by increasing the beam width for the legal nodes implementation to the point where the run-time roughly equaled the combination approach. In this situation it was also found that using a combination of the fitness functions still outperformed just the use of legal nodes.

TABLE 7
Performance of varying fitness functions for Beam Search

Dimension	Legal Nodes	Time(s)	Playouts	Time(s)
5	13	0.23	13	0.15
6	26	4.73	26	9.95
7	50	33.23	49	116.2
8	90	278.2	90	829.59
9	166	2043.6		

Dimension	Combination	Time(s)	Random	Time(s)
5	13	0.13	13	0.08
6	26	6.33	25	2.18
7	50	52.92	43	4.18
8	91	430.45	73	9.14
9	171	2951.3	124	22.75

Using Playouts alone produced shorter snakes and longer run times than both a combination and legal nodes functions. As predicted the purely random approach was clearly the worst performer, demonstrating that all other fitness function tested do offer significant heuristic improvements.

Despite the randomness present in the selection method, I found runs showed consistent performance between simulations in these tests.

The next aspect of Beam Search to test was the selection methods, Total Path and Reverse Tournament. These were again simulated with a 1000 beam width and we used the Legal Node fitness function for all further tests.

TABLE 8
Performance of varying selection methods for Beam Search

Dimension	Total Path	Time(s)	Reverse Tournament	Time(s)
5	13	0.19	13	0.23
6	26	3.22	26	4.73
7	50	19.44	50	33.23
8	88	175.6	90	278.2
9	163	1078.2.	166	2043.6

The two selection methods performed very similarly. However, the slight randomness in the Reverse Tournament selection proved to be slightly better at finding longer paths. Total path selection had much lower run-times, but I found this to not be worth the trade off in performance.

Priming was then implemented. This was done by running Beam Search to create a population of near optimal solutions at the lower level. This population was then used as a starting point for the higher dimension. To maintain consistency between tests, a beam width of 1000 was used. The results can be seen in Table 9.

The results showed significant increase in performance with the use of priming. The algorithm consistently returned longer snakes, and with a significantly reduced run-time.

TABLE 9
Performance of priming for Beam Search

Dimension	Unprimed	Time(s)	Primed	Time(s)
8	90	278.2	91	85.34
9	166	2043.6	174	646.2

The final aspect to investigate was the impact of Kochut's Constraint for canonical representation . All previous tests were conducted with the constraint and the results when this was removed were surprising. Whilst the lengths of the best snakes found decreased, we found surprisingly that the run-time also decreased in some cases.

TABLE 10
Performance of Kochut's Constraint for Beam Search

Dimension	With Constraint	Time(s)	Without Constraint	Time(s)
5	13	0.23	13	0.83
6	26	4.73	26	5.87
7	50	33.23	48	32.22
8	90	278.2	89	239.2
9	166	2043.6	160	1955.6

Our best implementation of Beam Search was now found, using a combination of fitness functions, reverse tournament selection, and both priming and Kochut's Constraint. Therefore we ran this implementation on longer runs/ beam widths, which took up to 8 hours to complete, to compare against best known results. It is worth noting that many of these record snakes took weeks, or even months, of core time to find [22], and so comparison with our results should be taken with that in mind.

The results are shown in Table 11.

TABLE 11
Best Performance of Beam Search

Dimension	Best Known Results	Beam Search
5	13	13
6	26	26
7	50	50
8	98	96
9	190	182
10	370	349

4.6 Nested Monte Carlo Search

Our first results for NMCS focused on the nested level of the search. Too many levels would result in too slow of a run-time, suffering the same issues as a breadth-first search, and too few levels lose the benifits of the nesting in the first place. For these testing simulations we arbitrarily chose the value of n, the probability of randomness in the biased search playouts, as 0.3. Each simulation was run for 180 seconds, and, due to the inherent randomness in the algorithm, each levels test was repeated 5 times and only the best performing result kept.

TABLE 12
Performance of Nested Monte Carlo Search for varying search levels

Dimension	level 1	level 3	level 5	level 7
5	13	13	13	13
6	26	26	26	26
7	47	48	50	46
8	87	87	88	83
9	152	155	149	146

The results in Table 12 tells us that, for the lower dimensions, using 5 levels of nesting provides the best results, even generating an optimal snake in dimensions 7. The performance of all levels fall off as the dimension increase, however it seems the higher levels are disproportionately affected.

It also appears that 7 nesting levels increases the search space too much, causing the algorithm to lower performance in comparison to lower levels.

Figure 7 displays the speed with which snakes of each length were found for each level during this testing. An interesting trend appears, lower levels tend to be quicker at finding the shorter snake lengths, but then take longer to extend their record found in comparison higher levels of search.

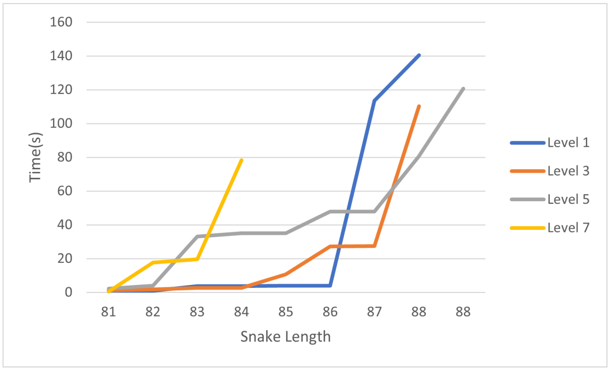

Fig. 7. Time taken to find Snake for varying search levels

Whilst the previous testing on Biased Search gives us some indication of the performance of the algorithm, this goes not necessarily convert into the performance of NMCS with different values of n with Biased Search as a playout method. Therefore we tested NMCS with different values of n, using 4 nesting levels.

TABLE 13
Performance of Nested Monte Carlo Search for varying n probabilities

Dimension	n = 0	n = 0.1	n = 0.3	n = 0.5
5	13	13	13	13
6	26	26	26	26
7	49	49	50	47
8	89	90	87	86
9	157	154	151	146

A value of n = 0, meaning no randomness present in the playouts, consistently produced long snakes and proved best in the highest dimensions. However, it was outperformed in dimensions 7 and 8 by 0.3 and 0.1 values respectively. It is worth noting that the randomness present in these cases caused variation in the longest snake found and these results were not repeated in every test, with n = 0 often outperforming over the short tests.

Overall, we determined 0.1 to be the most successful value for long runs.

Kochut's Constraint had little effect on NMCS. I found times to find snake of certain lengths marginally increased without canonical representation, but it was not significant.

On the other hand, once the difficulties of finding a strong priming point were overcome, priming proved very effective for NMCS, as shown in Table 14. Run were performed with n = 0.1, 4 levels of nesting, and left to run for 180s.

TABLE 14
Performance of priming for NMCS

Dimension	Unprimed	Primed
8	90	93
9	154	178

Finally, as with Beam Search, we ran our algorithm for 8 hours, using our best found variables: n = 0.1 and 4 nesting levels with priming and canonical representation. The results are shown in Table 15.

TABLE 15
Best Performance of NMCS

Dimension	Best Known Results	NMCS
5	13	13
6	26	26
7	50	50
8	98	96
9	190	180
10	370	342

4.7 Direct Comparison between algorithms

Table 16 shows a direct comparison between the best results found for Random Walk (RW), Biased Walk (BW), Depth-first Search (DFS) Beam Search (BS) and Nested Monte Carlo Search (NMCS).

Our longest snakes for every dimension have been found using Beam Search. Nested Monte Carlo Search has equalled these results up to dimension 8, and produced marginally worse paths for the higher dimensions. Both complex algorithms easily outperformed Random and Biased Walk in all cases.

4.8 Cherry-picked Primers

As previously mentioned, the results obtained whilst using cherry-picked primers can not be taken as solely our own

TABLE 16
Best Performance all algorithms for SIB

Dimension	Best Known Bounds	RW	BW	BS	NMCS
5	13	13	13	13	13
6	26	25	26	26	26
7	50	45	47	50	50
8	98	80	85	96	96
9	190	131	151	182	180
10	370	210	273	349	342

work. However, the results demonstrate how crucial the task of finding the best possible primer is in the generation of record breaking snakes, and provide us with insight into the potential performance of our algorithms if this is achieved.

Table 17 shows the results, all of which were found in under an hour of searching.

TABLE 17
Performance with cherry-picked priming snakes

Dimension	Best Known Bounds	Beam Search	NMCS
5	13	13	13
6	26	26	26
7	50	50	50
8	98	98	98
9	190	190	188
10	370	364	357

4.9 Coil-In-The-Box

The results for Coil-In-The-Box implementations followed our expectations closely. We found a direct correlation between algorithm performance for the SIB problem and performance in finding long coils. However the greater consistency of the Beam Search, in comparison with NMCS, combined with the population based approached seemed to translate into more consistency in the generation of long coils than NMCS.

Table 18 shows the longests coil found for Beam Seach and NMCS.

TABLE 18
Performance for Coil-In-The-Box

Dimension	Best Known Bounds	Beam Search	NMCS
5	14	14	14
6	26	26	26
7	48	48	48
8	98	93	90
9	188	176	171

5 Evaluation

5.1 Simple Search Algorithms

The performance of Random Walk and Narrowest Path Walk are roughly on par with expectations of the algorithms. Random Walk can produce impressively long snakes for the lower dimensions, given enough runs and thanks to the extremely fast run-time. However, the mean length is poor and it is clear that for dimension 8 and greater simple randomness is inadequate in generating competitive snakes in comparison with current records.

The performance of NPW is almost the opposite. Whilst the mean values are significantly better than Random Walk, it is clear that following this heuristic alone is not the best solution for creating long snakes. This is likely because although NPW maximises the number of available nodes, it also seems more likely to trap itself into a deadend following this approach.

The superiority of Biased Search over both Random Walk and NPW is significant. The mean path length found was still lower than that of NPW for all values of n, and especially lower for larger n values. However the longest snakes found significantly outperformed both previous algorithms for every dimension. The finding that low values of n, despite performing better for high dimension, generated shorter paths for dimensions 6 and 7 further indicates that more randomness is vital for small searcher spaces, but becomes detrimental for larger search spaces.

The most surprising find was that setting a value of 0.9 for n decreased performance in comparison to RW. We suggest that this is a consequence of the higher chance of deadends by following the NPW heuristic, with too much randomness to benefit from the advantages of it.

BFS and DFS do an excellent job demonstrating the combinatorial explosion of the Snake-In-The-Box problem, with the exhaustive algorithms only able to find optimal solutions for the first 5/6 dimensions respectively without any optimisations.

Both graph theory optimisations, Kochut's Constraint and Priming, were shown to have massive impacts on performance and can be viewed as essential when attempting to find competitive snakes for high dimensions.

5.2 Beam Search

5.2.1 Solution Strengths and Observations

Overall, the performance of Beam Search for the SIB problem is impressive. The use of a combination of the Legal Nodes heuristic and playouts for the fitness function clearly outperforms either one individually, demonstrating that it does succeed, at least in part, at negating the flaws of each respective approach. The increased computational complexity, and therefore longer run-time of this approach is shown to be worth it for the performance increase in generating longer snakes.

Similarly, we found the increased run-time of Reverse Tournament Selection to be a worthy trade off for performance in comparison to Total Path Selection. We were expecting RTS to increase variance in results, with different runs returning significantly different length snakes (both higher and lower than TPS). However we were surprised to find the results remained consistent, and reliably better than that of Total Path Selection. This demonstrates that the randomness increases the chance for a heuristically worse path to survive pruning, without increasing the chance of a heuristically very strong path being pruned.

Priming was a notable strength of Beam Search. Not only did it increase performance of snakes found and reduce run-time, but it was also far easier to implement for Beam Search in camparison with other algorithms. This is a consequence of Beam Search taking a population of snakes as an input, rather than just a single path.

Another unexpected result was the effect of removing Kochut's Constraint. The generated snake lengths decreased, and run-time actually decreased, which was a surprise as one would expect the increased search space without canonical representation to increase paths found at each level and hence increase run-time. This finding must therefore be attributed to more poor paths not being pruned, and hitting deadends. The reduced path lengths are likely caused by many "equivalent" paths taking up places in the beam width and reducing diversity. Overall, as expected, Kochut's Constraint proved beneficial for Beam Search performance.

When running with our optimal setup, the results in comparison with the known records were extremely encouraging. We were able to find paths for dimensions 1-7 very consistently and quickly ($<$ 30s for dimension 7) and maintain respectable scores for higher dimensions, despite significantly shorter run-times than that of paths record setting algorithms.

5.2.2 Solution Weaknesses

We did not find Beam Search to have many glaring weaknesses. However, one drawback is that it is not an "anytime algorithm" meaning to generate a complete snake, the algorithm must be run to completion. We had no indication of the final snake length found at any point before the finished search.

5.3 Nested Monte Carlo Search

5.3.1 Solution Strengths and Observations

We found that the optimal nesting level for Nested Monte Carlo Search differed depending on both the dimension and the length of time the algorithm was left to run. Fewer levels tended to find lower lengths snakes faster, and performed better for higher dimensions than higher levels. However higher levels found longer snakes faster (as depicted in figure 5) and level 5 was found to perform best for dimension 7 and 8.

This is likely due to the trade off in increased computational complexity that additional nesting makes. Nesting levels higher than 5 were shown to negatively impact performance in all cases. Nevertheless, nesting did prove effective and 4 levels was found to be a sweet spot for most dimensions.

The optimal value for n in our biased playouts seemed to behave in the same way as for the previously mentioned Biased Walk. Larger amounts of randomness proved beneficial in the lower dimensions, but when the search space was greater in the higher dimensions, it proved to be detrimental to performance.

We also observed little impact with the use of Kochut's Constraint. This is neither a strength nor a weakness, but demonstrated that in comparison to beam search and any exhaustive method, NMCS is far less dependent on the search space size.

5.3.2 Solution Weaknesses

Although priming significantly increased the performance for NMCS, this is also the largest weakness of the algorithm. The large dependence on using primed starting points, combined with the difficulty of finding an effective primer, is a huge negative in comparison with Beam Search, which is both less reliant on the method, and much easier to use with priming.

We found consistency to be an issue for the Monte Carlo approach, likely due to the significant level of randomness present. Taking dimension 7 as an example, optimal solutions of length 50 were, in some cases, found in under 150s without priming. However, on other runs the same algorithm could take up to 1500s to find this solution, a more than 10x increase in run-time.

5.4 Cherry-picked Results

The results of the implementations incorporating cherry-picked priming demonstrate clearly the potential of our solutions and the huge impact of finding the correct path to prime the algorithms, which as previously mentioned is an extremely difficult task.

5.5 Comparison

On the whole, Beam Search has proven to be the superior approach to NMCS in most cases. It has found longer snakes given the same time period for all longer runs, and has proven much easier to implement and run effectively than NMCS.

The only obvious advantage for the Monte Carlo Search lies in the nature of being an anytime algorithm, allowing it to be stopped at any point, or left to run indefinitely to produce better and better results. Conversely, Beam Search must fully complete the algorithm to return snakes. For this reason, NMCS was found to be more effective in the specific scenario of extremely short runs, especially for higher dimensions.

6 Conclusion

We found Beam Search to be our most effective tested algorithm. Using our best found implementation setup we were able to construct optimal snakes for the first 7 dimensions, and competitive lengths for dimensions 8-10. Nested Monte Carlo Search was not far behind in performance, however lacked consistency between runs, and relied heavily on priming. Both algorithms clearly outperform our more simple implementations and the exhaustive options available.

The strong performance of Beam Search does not come as a particular surprise, given its excellent performance in previous implementations. Performance of NMCS did overall fall slightly below expectations given it also is a record holding algorithm (for dimension 10).

In terms of meeting our deliverable, the project was largely a success. We were able to develop a considerable understanding of the Snake-In-The-Box problem, and implement a number of both simple and more complex algorithms to solve it. These were extensively tested to find the optimal parameters in order to maximise the length of snakes found. The impact of graph theory approaches were measured and proved essential in scaling results up to higher dimensions of the problem. We were unable to meet all advanced deliverables, specifically breaking known records (without cherry-picked primers), however given time constraints we were only able to test with short simulations and hence chose to focus more on experimentation. We believe that our approach to the Beam Search algorithm has shown to have significant potential.

6.1 Future Work

Following on from our work, the next steps would be to build the algorithms in a more efficient language, for example C++, making use of parallelisation and experimenting with longer runs on more powerful computers. This would serve as a better like to like comparison against record holding algorithms.

It would also allow for more experimentation into performance in the higher dimensions of the problem. We theorise that heuristic approaches alone are insufficient as solutions for $m > 14$ and hence more work needs to be done in the combination of heuristic and graph based methods if longer solutions are to be found.

References

[1] Rashmi K A and B Kalpana. A genetic algorithm for the snake-in-the-box problem using modified edge recombination. *Procedia Computer Science*, 165:642–646, 2019. 2nd International Conference on Recent Trends in Advanced Computing ICRTAC -DISRUP - TIV INNOVATION , 2019 November 11-12, 2019.

[2] H.L Abbott and M Katchalski. On the snake in the box problem. *Journal of Combinatorial Theory, Series B*, 45(1):13–24, 1988.

[3] D. Allison and D. Paulusma. New bounds for the snake-in-the-box problem. Working paper, DU, September 2019.

[4] Kartheek Atluri. Snake-in-the-box problem using nature inspired search, 2009.

[5] Mohammed Ayar. Python lists are sometimes much faster than numpy. here's proof., Apr 2021.

[6] Derrick Scott Bitterman. New lower bounds for the snake-in-the-box problem: a prolog genetic algorithm and heuristic search approach, 2004.

[7] Benjamin P. Carlson and Dean F. Hougen. Phenotype feedback genetic algorithm operators for heuristic encoding of snakes within hypercubes. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, page 791–798, New York, NY, USA, 2010. Association for Computing Machinery.

[8] Darren Casella and Walter Potter. New lower bounds for the snake-in-the-box problem: Using evolutionary techniques to hunt for snakes. pages 264–269, 01 2005.

[9] Tristan Cazenave. Nested monte-carlo search. In *IJCAI*, 2009.

[10] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed optimization by ant colonies. 01 1991.

[11] Carnegie-Mellon University.Computer Science Dept. Speech understanding systems: summary of results of the five-year research effort at Carnegie-Mellon University. 4 2015.

[12] Marco Dorigo and Luca Maria Gambardella. Ant colonies for the travelling salesman problem. *Biosystems*, 43(2):73–81, 1997.

[13] Michael C. Fu. Alphago and monte carlo tree search: The simulation optimization perspective. In *2016 Winter Simulation Conference (WSC)*, pages 659–670, 2016.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.

[15] Shilpa P Hardas. An ant colony approach to the snake-in-the-box problem. 2005.

[16] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. second edition, 1992.

[17] W. H. Kautz. Unit-distance error-checking codes. *IRE Transactions on Electronic Computers*, EC-7(2):179–180, 1958.

[18] D. Kinny. A new approach to the snake-in-the-box problem. *Frontiers in Artificial Intelligence and Applications*, 242:462–467, 01 2012.

[19] Victor Klee. What is the maximum length of a d-dimensional snake? *The American Mathematical Monthly*, 77(1):63–65, 1970.

[20] Krys J. Kochut. Snake-in-the-box codes for dimension 7. *Journal of Combinatorial Mathematics and Combinatorial Computations*, 20:20–175, 1996.

[21] Blaum M and Etzion T. Use of snake-in-the-box codes for reliable identification of tracks in servo fields of a disk drive, 2002.

[22] S. J. Meyerson, W.E. Whiteside, T.E. Drapela, and W.D. Potter. Finding longest paths in hypercubes, snakes and coils. In *2014 IEEE Symposium on Computational Intelligence for Engineering Solutions (CIES)*, pages 103–109, 2014.

[23] Seth Meyerson, Thomas Drapela, William Whiteside, and Walter Potter. Finding longest paths in hypercubes: 11 new lower bounds for snakes, coils, and symmetrical coils. 06 2015.

[24] Patric R.J. Östergård and Ville H. Pettersson. On the maximum length of coil-in-the-box codes in dimension 8. *Discrete Appl. Math.*, 179(C):193–200, dec 2015.

[25] Walter D. Potter, Robert W. Robinson, John A. Miller, Krys J. Kochut, and D. Z. Redys. Using the genetic algorithm to find snake-in-the-box codes. In *IEA/AIE '94*, 1994.

[26] F. Preparata and J. Nievergelt. Difference-preserving codes. *IEEE Transactions on Information Theory*, 20(5):643–649, 1974.

[27] PyPy, Dec 2019.

[28] Daniel Tuohy, Walter Potter, and Darren Casella. Searching for snake-in-the-box codes with evolved pruning models. pages 3–9, 01 2007.

[29] Ed Wynn. Constructing circuit codes by permuting initial sequences, 2012.

[30] Yiwei Zhang and Gennian Ge. Snake-in-the-box codes for rank modulation under kendall's τ -metric in s_{2n+2}. *IEEE Transactions on Information Theory*, 62(9):4814–4818, 2016.

[31] Igor Zinovik, Daniel Kroening, and Yury Chebiryak. Computing binary combinatorial gray codes via exhaustive search with sat solvers. *Information Theory, IEEE Transactions on*, 54:1819 – 1823, 05 2008.