Nathan Comer
4511w Peer Response

Paper by: Dalton Wiersma

Answer the following questions:

1. What do you take as the focus or main point of this draft?
The benefits and drawbacks of using Backtracking Depth First Search over Breadth First Search and Depth First Search.

2. What, specifically, interested you about the draft?
The explanations of the differences between the algorithms interested me. The paper does a good job comparing and explaining the differences between the algorithms and explaining how the algorithms work in words.

3. What do you suggest as the single most important revision your peer could make?
Adding quantifiers to explanations and ideas.
For example:
-give the mathematical analysis backing your explanations
-provide details in a manner meant more for a technical audience

Also providing data in a manner that shows the contrasts between the algorithms.
For example:
-side by side tables
-graphs

For the following,
RATE 1 to 5 (5 is high):

4 How engaging and useful did you find the Introduction? Any suggestions?

3 Audience: Presentation is professional and details provided are for a technical audience.

4 Clarity: Provides clear explanation of project to reader. Reader can understand motivation, experiments, analysis, and results.

4 Cohesion and Focus: Write-up overall presents a single focused project. Sections and paragraphs have clear topics that are supported and developed within that section or paragraph.

4 Fluidity and Transitions: Write-up unfolds a “story” to the reader about project experiments. Organized and ordered logically. Ideas transition between paragraphs and sections.

3 Formatting: Write-up has good style with respect to creativity and formatting. Project presentation is engaging. Formatting makes document look professional and helps reader navigate document.

Nathan Comer
4511w Peer Response

Paper by: Jacob Rice

Answer the following questions:

1. What do you take as the focus or main point of this draft?
Comparing BFS and Backtracking search (uniformed search algorithms) on KenKen puzzle and Cryptarithmetic.

2. What, specifically, interested you about the draft?
The paper does an excellent job making an algorithm as simple as bfs interesting. It explained the purpose of uninformed search in an interesting and convincing way. Most importantly the paper provides concise, yet thorough, explanations. Results and details are presented in a technical manner consistent with the target audience.

3. What do you suggest as the single most important revision your peer could make?
I suggest spending more time on data collection. The data you have is explained and analyzed extremely well, but having more, and a greater variety, of data would make the paper more interesting. You also discuss how BFS and Backtracking are not necessarily the best algorithms for KenKen and Cryptarithmetic. Perhaps adding a 3rd puzzle, one that BFS and Backtracking are relatively good at, would add a real world motive to the paper.

Breaking some of the larger sections of the paper into smaller sections, or grouping some of the information into subsections may help the flow of the paper.

For the following,
RATE 1 to 5 (5 is high):

5 How engaging and useful did you find the Introduction? Any suggestions?

5 Audience: Presentation is professional and details provided are for a technical audience.

5 Clarity: Provides clear explanation of project to reader. Reader can understand motivation, experiments, analysis, and results.

4 Cohesion and Focus: Write-up overall presents a single focused project. Sections and paragraphs have clear topics that are supported and developed within that section or paragraph.

4 Fluidity and Transitions: Write-up unfolds a "story" to the reader about project experiments. Organized and ordered logically. Ideas transition between paragraphs and sections.

5 Formatting: Write-up has good style with respect to creativity and formatting. Project presentation is engaging. Formatting makes document look professional and helps reader navigate document.

Backtracking Depth first Search versus Breadth first search

Dalton Wiersma
College of Science and Engineering
University of Minnesota
Minneapolis, Minnesota
Email: wiers043@umn.edu

Abstract—**Backtracking Depth First Search (BDFS) is a search algorithm that explores states of a problem in a very similar manner to traditional Depth First Search (DFS). Unlike DFS, BDFS does not generate new states at every node in the search tree, instead saving a single state, which is modified by applying and undoing single actions that are stored within nodes. This paper presents the time and memory benefits and drawbacks of using BDFS over Breadth First Search (BFS) and DFS.**

I. Introduction

The main benefit of using a traditional DFS over a BFS is the drastic improvement in memory usage. While BFS traverses a tree, it explores every node at a particular depth before moving on to the next level. As a result of this, every node at any particular depth must be placed in the queue used by the algorithm before any single node at that depth is explored. This means that for any given problem, the maximum memory used at any point is the total number of of nodes at the depth where the soltuion lies if that solution is at maximum depth. If there is no maximum depth for the problem, the amount of memory could be even larger.

I would expand on this more being
it's one of the biggest advantages
BFS has over the other algorithms discussed

BFS guarantees that any solution found is an optimal solution, since it will always find the solution at the lowest possible depth. DFS and BDFS do not share this trait, and generally only work with problems that have a maximum depth, otherwise it will follow a single branch of the tree until it finds a solution, or until there are no more possible actions to be applied. Because of this, BDFS and DFS are limited to certain problems, Where BFS can be applied more universally. A workaround for this would be to use an Iterative Depth - Depth First Search, which limits and gradually increases the depth allowed for the search.

In this paper I explore the the application of these algorithms on the n Queens problem and the Cross Math problem. These two problems are similar in nature, both in the process that must be gone through to solve and how a given state is represented. Both of these problems will be explained in greater detail in a later section.

II. Breadth First Search

```
1   def BFS(problem):
2    node = Node( problem.initial )
3
4    if problem.isGoal( node.getState() ):
5      return node
6      frontier=deque()
7      frontier.append(node)
8
9    while len(frontier) > 0:
10     node = frontier.popleft()
11     for child in node.expand(problem):
12       if problem.isGoal( child.getState() ):
13         return child.getState()
14       frontier.append(child)
15   return None
```

Be sure that in the final draft the algorithms are not cut in half

Adding equations (big O, runtime, memory use, etc)
may help to improve the explanations of
the differences between the algorithms

Algorithm 1: Breadth First Search. BFS searches through the problem state by generating an intial state and encapsulating it within a node. The function expand (line 11) is called on this node, which returns a series of nodes containing the states resulting from every possible action from the previous state. These node are then placed in a queue, which is then polled, restarting the process. The biggest drawback to this algorithm is the memory required. Since every possible state at any given depth can be in the queue at once, the algorithm can demand an excessive amount of memory for large problems. The memory complexity cannot be reduced by saving a single state and simply using nodes to save a single action. This is because each state explored does not necessarily have anything in common with the previously explored state. As a result, if a large number of actions need to be undone, there is no way for it to know which actions must be undone, unless each node stores every action needed to get to that state. However, this would prove to be counterintuitive, as this list of actions could quickly grow to become greater than the size of the state. A benefit of this algorithm over others is it can be applied to a large majority of problem types, and is always guaranteed to find a solution at the minimum possible depth.

<----
Good
explanation

Breadth first search also provides a framework on which many other algorithms are based. One of these algorithms is A star search, which works by taking every state at a given depth and sorting them based on a heuristic function. They are given a value corresponding to how ”promising” the heuristic determines them to be. They are them placed in a priority queue, which is then drawn from to receive the next state [2]. This use of a priority queue over a traditional queue is a rather small change, but it completely changes the search path, becoming much more directed than its predecessor.

Organizing the steps the algorithm goes through in a format such as a list may improve this explanation, current paragraph is long and somewhat hard to follow

III. Backtracking Depth First Search

Backtracking DFS explores states in the same order as a traditional DFS. It works by taking all possible actions from the current state, encapsulating those actions and the depth of the state that would result form applying that action, and placing that node on a stack. When that node is popped off of the stack, its action is applied to the current state, after which the process will start over. The depth is stored in each node, as well as the current depth of the state. The purpose of the is to determine the number of actions that must be undone before the current action is applied. The actions that have been applied to the current state are stored in a seperate stack, and when an action must be undone, it is popped off the stack and passed to the undo method of the problem class. While this method of graph traversal may seem ideal in many cases, there are many limitations as well. Because the latest information must always be known to make any further computations, the depth-first approach in particular is considered ”a nightmare for both parallel processing and memory hierarchies” [1].

?

?

The benefit of using this implementation of Depth First Search over a more traditional one stems from the the size of the nodes. While the main benefit of traditional DFS is how little memory it requires when compared to its BFS counterpart, Backtracking takes it one step further. Depending on the problem being solved, the amount of memory required to store an entire state can be significant. Also, the sequence of states that would be stored at any given time are often very similar to one another, with each one only slightly different than the previous. Backtrackng DFS takes advantage of this fact by only storing the original state, and the series of changes to be made to that state to reach a possible solution.

```
 1 def BDFS(problem) :
 2   acts = problem.getActions( problem.getState() )
 3
 4   frontier=deque()
 5
 6   for act in acts:
 7     frontier.append( Node(act, 0))
 8
 9   maxNodes = 0
10   currentDepth = 0
11
12   actionStack = deque()
13
14   while len(frontier) > 0:
15     node = frontier.pop()
16     if len(frontier) > maxNodes:
17       maxNodes = len(frontier)
18
19     while currentDepth > node.getDepth():
20       actionNode = actionStack.pop()
21       problem.undoAction( actionNode.action )
22       currentDepth -= 1
23
24     problem.applyAction( problem.getState(), node
       .action )
25     actionStack.append( node )
26     currentDepth += 1
27
28     if problem.isGoal( problem.getState() ):
29       return [maxNodes, node.getNodeCount()]
30
31     for child in node.expand(problem):
32       frontier.append(child)
33
34   return None
```

Algorithm 2: Backtracking Depth First Search. In line 31, the ”expand” function refers to a method in the node class that returns all the possible actions from the current state, each of them in a new node.

Consider turning each puzzle section into two sections. For example: 1. Explanation and Analysis 2. Results

IV. N Queens

The N Queens problem is a fairly straightforward one. For an n x n chess board, you must place n ”queens” on the board. Each piece must not share a row, column, or diagonal with another piece. Pruning for the problem was done inside the functions that generates actions, checking that the placement of the next piece satisfied the constraints of the problem.

Not a great way to introduce this section

TABLE I
Performance of N Queens with BDFS - Pruning Implemented

Size	Time	Maximum Memory
4	17	6
5	38	10
6	214	15
7	396	21
8	2028	28
9	5599	36
10	24372	45
11	121150	55
12	765978	66
13	5191412	78
14	41219093	91

These tables are great for showing the data but not so great for comparing data between the algorithms. Consider adding a 3rd table or a graph that compares the results.

TABLE II
Performance of N Queens with BFS - Pruning Implemented

Size	Time	Maximum Memory
4	26	11
5	121	39
6	528	159
7	2476	743
8	13149	3967
9	78134	23715
10	516129	157463
11	3743341	1142599
12	29598156	9042415

The time metric for this problem was measured in total nodes created. This was measured by incrementing a global variable with every call of the node constructer function. Time was measured in this way to provide a more accurate measurement of the amount of processing needed to solve the particular problem give. The alternative was using a

Great explanation for why you chose this metric more standard time measurement that can be impacted by the processing power of the machine used to run the experiment, as well as the state of the machine at any given time that might also skew the data. The maximum memory was determined by the monitering the size of the stack or queue and saving the highest number of nodes in it at any given time.

BDFS greatly outperformed BFS for this specific problem. BFS did not hold any advantages in either memory or time complexity. The maximum memory for BFS grew exponentially with increased problem size while BDFS only grew linearly with the maximum depth of the problem. This is because BDFS only expands one node at a given depth at a time, which is drastically different than how BFS expands every single node at a given depth before moving on. In addition to this large disparity, the size of the nodes used by BDFS are sigificantly smaller than those used by BFS, since they only store two numbers, the next number to be placed in the state and the depth of that node in the search tree, where BFS must store the entire state.

Pruning for this problem was implemented when a state was promted by the algorithm to generate its child states. Before the generating these possiblities, it was first checked that the most recent addition to the state does not conflict with any of the previous additions. If there are any conflicts, then it can be ruled out that any child of that state could possibly be a solution. As a result, no children will be generated.

TABLE III
PERFORMANCE OF N QUEENS WITH BDFS - NO PRUNING

Consider adding this as a caption

Size	Time	Maximum Memory
4	19	6
5	62	10
6	456	15
7	2406	21
8	31284	28
9	191432	36
10	1257432	45
11	21916638	55
12	160495216	66

Adding graphs would make some of this data easier to understand

TABLE IV
PERFORMANCE OF N QUEENS WITH BFS - NO PRUNING

Size	Time	Maximum Memory
4	28	24
5	162	120
6	906	720
7	8543	5040
8	71365	40320
9	697361	362880
10	9642938	3628800

Without pruning, N Queens, like any other problem, sees a huge drop in performance. The total number of possible states at a given depth N is N factorial, so without pruning to eliminate most branches of the search tree, the time required to solve the problem can quickly become hours before N approaches 8. When comparing the results when using pruning versus no pruning, it becomes clear how important it can be. When solving large problems, pruning can be the difference in hours of computing.

Should add this equation in up above where it is discussed

$$\sum_{i=1}^{n} i! \tag{1}$$

V. CROSS MATH

	+		÷		=	9
+		÷		+		
	×		+		=	26
−		+		−		
	+		+		=	16
=		=		=		
8		10		4		

add parts of this explanation as a caption for the puzzle image

The cross math puzzle is configure as shown above, and the goal is to fill each empty box with unique numbers 1 through n, where n is the total number of empty boxes, while each row and column evaluates to the numbers in the far right column and bottom row.

TABLE V
PERFORMANCE OF CROSS MATH WITH BDFS

Size	Case	Pruned	Time	Maximum Memory
2	Best	No	6	6
2	Worst	No	66	6
2	Average	No	13.4	5.2
2	Best	Yes	FILL	FILL
2	Worst	Yes	FILL	FILL
2	Average	Yes	13.4	5.2
3	Best	No	102	36
3	Worst	No	986502	36
3	Average	No	2243.0	27.2
3	Best	Yes	FILL	FILL
3	Worst	Yes	FILL	FILL
3	Average	Yes	2243.0	27.2

The metrics used for time and memory complexity are the same used for the N Queens problem. These metrics are time measured in total nodes created and memory measured in maximum nodes stored at one time. The best and worst cases were determined by which node at the depth of the solution would be explored first or last at that depth. The best case for BDFS, for example, was [1, 2, 3, 4, 5, 6, 7, 8, 9], since that

TABLE VI
PERFORMANCE OF CROSS MATH WITH BFS

Size	Case	Pruned	Time	Maximum Memory
2	Best	No	41	23
2	Worst	No	105	23
2	Average	No	52.1	23
2	Best	Yes	FILL	FILL
2	Worst	Yes	FILL	FILL
2	Average	Yes	23.4	11.2
3	Best	No	623635	362879
3	Worst	No	1610044	362879
3	Average	No	800615.3	362879.0
3	Best	Yes	FILL	FILL
3	Worst	Yes	FILL	FILL
3	Average	Yes	4020.5	1991.2

traversal pattern will always explore the action with the lowest remaining number first. The worst case would be the reverse order of the best case for the same reason. The average case was determined by solving 1000 randomly generated Cross math puzzles of each size, then determining the averages of the time taken and the maximum amout of memory used. BDFS had a widely varying amount of processing time required to solve between the best and worst case when the state size became large. This was expected because the best case will be found after exploring one state at each depth, where the worst case must explore every possible state before reaching the correct one.

BFS showed a wide variance between the best and worst case as well, however it required exponentially more computation before reaching the first possible solution. The worst case took 2.58 times longer than the best case, while the worst case took more than 9000 times more when using BDFS. Since BFS solve times grow exponentially with increasing problem size, it appears to be nearly unuseable for any problem with a large state size. This drawback is greatly reduced when pruning is introduced. For the Cross Math problem, pruning was implemented at a very basic level. When the algorithm prompted a Cross Math state for its children states, it first checked if a row of the problem had just been filled with the most recent action. If it had, it would evaluate that row and ensure that the result was equal to the desired result for that row. If these two values don't match, then there will be no children produced from that state, since there are no possible correct solutions that can come from it. This small check results in a significant reduction of the state space. The effect of pruning is so prominent because it becomes more powerful with an increased state size. For example, for a two by two puzzle, the same 1000 puzzles that were solved with an exploration of an average of 52 nodes without pruning, was solved in less than 45 percent of the time with the use of pruning. When solving a three by three puzzle, BFS needed an average of over 800,000 nodes created before it reached a solution. By comparison, the implementation of pruning resulted in the average time to solve being nearly 200 times less.

Great use of quantifiers

VI. CONCLUSION

Based on the results from the two previous experiments, Backtracking Depth First Search appears to reach a solution in less time while using less memory than Breadth First search. However, these results only apply to the specific type of problem that were solved. This problem type is defined by the solution always being found at specific known depth. When used to solve problems in which actions can be applied to a state infinitely, BDFS will be susceptible to falling into infinite loops within the state graph, while BFS may quickly run out of memory, however it is guaranteed to find an optimal solution for any of these problems when not constrained by the memory capacity of the machine. The Breadth First Search algorithm has many drawbacks, but when used to solve a problem that implements pruning, it can can be a viable option in many situations. The dramatic performance improvements that can be observed when implementing pruning also show that it can provide a very useful framework for which new search algorithms can use.

REFERENCES

[1] Kurt Mehlhorn and Peter Sanders, *Algorithms and Data Structures: The Basic Toolbox*, 3rd ed.   Karlsruhe, Germany: 2007.

[2] Daniel Delling, Peter Sanders, Dominik Schultes, Dorothea Wagner, *Engineering Route Planning Algorithms*,   Springer. Berlin, Heidelberg, Germany: 2009.

[3] Even, Shimon, *Graph Algorithms*, 2nd ed.   Cambridge University Press. Karlsruhe, Germany: 2007.

Consider also exploring variations of bfs, or algorithms similar to bfs, or adding in more different pruning techniques

Uninformed Search On Large Datasets

Jacob Rice
ricex602@umn.edu

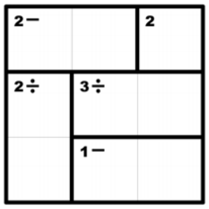
***Abstract*—The efficiency of Breadth-First Search (BFS) and Backtracking Search were compared in solving KenKen and Cryptarithmetic. KenKen puzzles between the sizes of 4×4 and 8×8 were used, and Cryptarithmetic problems were generated with random solutions, each puzzle using between 4 and 19 words. Execution times and number of nodes held in memory were measured during execution. Backtracking was found to run in approximately the same time as BFS, and it held very few items in memory, while BFS used significantly more memory.**

Could add another paragraph to the introduction instead of jumping right into the puzzles

I. Introduction

KenKen is a mathematical puzzle in which groups of cells, called *cages*, must evaluate to a defined number, using the digits 1 to n only once in each row and column on an $n \times n$ board [3]. It is similar to Sudoku in many ways, but the introduction of arithmetic leads to an interesting difference in terms of computing. An example puzzle is shown below:

Good concise explanations of puzzles



Source: kenkenpuzzle.com

Cryptarithmetic is an arithmetic puzzle in which digits are represented as letters. The goal is to find what digit each letter must represent in order to solve the long-arithmetic problem. Problems are typically represented as words or sentences. An example puzzle is shown below:

$$\begin{array}{r} \text{S E N D} \\ +\,\text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$

Source: wikipedia.org

This paragraph does a great job introducing the paper and explaining the importance of the topic

Both problems can be solved with more advanced artificial intelligence techniques, such as Constraint Satisfaction, but uninformed search methods have many other practical applications for which these problems provide a useful comparison. Unorganized, multi-level directory structures, for example, are a structure in which uninformed search techniques are not only useful, but are required.

A. Breadth-First Search

Breadth-First Search, or BFS, searches every node at a certain level in a tree before exploring each of those nodes' children. In doing so, it must maintain a complete list of every unexplored node, which greatly increase its space requirements in comparison to other search algorithms. BFS has the benefit of quickly finding consecutive solutions on constant-depth trees, such as those created by a Sudoku board, and can return multiple solutions if they exist in a similar time to a single return. In addition, numerous studies have been conducted about parallel graph traversal using multiple cores with great success [1].

```
## Simplified BFS code
def BFS(problem):
  node = problem.initial
  if node.isGoal():
    return node
  frontier = queue(node)
  while len(frontier) > 0:
    node = frontier.pop()
    for child in node.expand():
      if child.isGoal():
        return child
      frontier.append(child)
  return None
```

B. Backtracking

Good concise explanation

Backtracking Search is a depth-first traversal in which only a single node and its children are held in memory at any given time, and traversal back up the tree is done by a defined "undo" action, which effectively returns the problem state to the level above. With this method, memory requirements are minimal, and runtime is comparable to Depth-First Search.

```
## Simplified Backtracking code
state = None
def Backtrack(problem):
  if backtrackSolve(problem):
    return state
  return None

def backtrackSolve(problem, node):
  state = node
  if state.isGoal():
    return True
  for action in state.getActions():
    actions.append(action)
    node = makeNode(action)

    if backTrackSolve(problem, node):
      return True

    state = undoAction(state)
    actions.pop()
    node = state
  return False
```

C. Theoretical Analysis

1) BFS: Because both KenKen and Cryptarithmetic have predefined numbers of variables, the search trees for each will have constant depth d. However, due to the pruning techniques, subtrees with no valid next steps are removed, giving the tree an average depth d'. The maximum tree for an $n \times n$ KenKen puzzle will have depth $d = n \times n$, and the tree for Cryptarithmetic will have a depth equal to the number of unique letters used in the problem, limiting it to a depth of $d = 10$.

The branching factors b for these puzzle vary widely based on the number of variables already assigned, the numbers and operators in KenKen cages, and the number and length of words in Cryptarithmetic problems. The aggressive pruning employed in these experiments ensure that the branching factor decreases consistently as the search depth increases, so the factor will be approximately half of the inital branching factor, $b = b_{\text{initial}}/2$.

BFS visits every node in a tree until the goal is found. Therefore, the expected runtime of BFS is

Good expalation of runtime

$$E[t_{\text{BFS}}] \approx b^{d'} - 1 + \frac{b^{d'}}{2} \tag{1}$$

since there are approximately $b^{d'}$ nodes at the deepest level of the tree, and the expected location of the goal is halfway across that level [2].

Since BFS holds all of its nodes in memory, it will have an expected space efficiency equal to the time efficiency defined above.

2) Backtracking: Backtracking has the same expected runtime as Depth-First Search. Its runtime varies widely depending upon the location of the goal in the final level of the tree, meaning it can range anywhere from d to b^d. Since the tree has constant depth, the approximate expected runtime of Backtracking is

$$E[t_{\text{Backtrack}}] \approx \frac{b^{d'+1} - 1}{2} \tag{2}$$

because with an expected goal location halfway across the deepest level, the depth-first algorithm will only need to traverse half of the nodes on average.

Because it only holds actions in memory, it will only hold d items at a time.

II. Experimental Set-Up

Great explanation of metrics used and why

In order to measure the number of nodes encountered during execution, node counters were implemented inside of each of the search functions. A timing function surrounded the code execution in order to note the algorithm's search time in seconds. These two metrics can be compared to form a more meaningful concept of execution time, with the former ignoring inconsistencies in the CPU time of the function, and the latter giving an applied, experimental value for the search time. All tests were run on the same machine to minimize differences in execution time.

Measurements of the amount of memory used by each function was implemented by measuring the maximum number of nodes or actions of which each algorithm kept track at once. In BFS, a list of nodes, called the `frontier`, held the unexplored nodes. In Backtracking, a list `actions` was kept, representing edges between nodes of each level. Explicit RAM measurements were not taken due to the small size of nodes and the relatively large memory requirements of Python. All data was written to CSV files during execution.

Both algorithms were tested on 5 KenKen problems with sizes between 4×4 and 8×8. These examples were randomly generated online, so each had random difficulties compared to the other tests. Each of the puzzles made use of addition, subtraction, multiplication, and division operators in the cages, and each cage consisted of 4 or fewer cells. Because KenKen's complexity scales so sharply with sidelength, differences in difficulty corresponding to cage operations or layouts were assumed to be minimal.

The algorithms were also run on 80 randomly-generated Cryptarithmetic problems of varying sizes, twice each to counter any anomalous execution times. The number of words in each test ranged from 4 to 19, with average word lengths ranging from 3 to 24 characters. Analysis of a puzzle's difficulty based on average number of letters per word showed a negligable correlation to execution time, so 10 puzzles with varying word lengths were generated for each of the 4- to 19-word puzzles.

To generate a puzzle of length ℓ, random numbers between 10^{w-1} and 10^{w} for a random $w \in [3, \ell]$. A random operator was chosen and applied to the numbers, and the result was recorded. Random letters were then assigned to the numbers and mapped to each digit, and the obfuscated puzzles were saved to a file in order to be run with both Backtracking and BFS.Good explanation of how puzzles were generated and tests were performed

III. Results and Analysis

A. KenKen

As KenKen boards increase in size, they also increase in complexity. In this implementation, actions correspond to the filling of a cage, so a cage of size c cells will have at least $c!$ initially possible states, with more considering different possible number combinations. For example, on a 8×8 board, a 3-cell cage with operator $+$ and answer 8 could initially have solutions $\{1, 2, 5\}$, $\{1, 3, 4\}$, $\{2, 3, 3\}$, and so on. Pruning techniques for this implementation eliminate repeated numbers in rows and columns, but, early in the search, little information is known about the problem, so large problems start with a large branching factor, with an upper bound of $C!$, where C is the average cage size.

BFS and Backtracking execution time and storage space were evaluated first for KenKen problems of sizes 4×4 to 8×8. **Figure 1** shows the runtime with BFS of the KenKen test problems, and it yields a best-fit exponential approximation with $R^2 = 87.58\%$. This exponential function is approximated because although the tree is initially factorially expanding initially, information in the rows and columns grows quickly and restricts the possibilities at a linear rate, dramatically decreasing the rate of growth. For this small data set, the rate of growth can be treated as approximately exponential.

Consider breaking these long sections into smaller subsections
(maybe one subsection per graph)

BFS checks every node at a given depth before exploring their children. In the case of KenKen, it fills each cage with all of its possible combinations and permutations before moving to the next. With this method, it wastes a significant amount of time if the solution is near the "middle" of the tree, as it will explore half of the tree that it does not need to explore. It will have a more consistent and larger search time than Backtracking because of this.

Throughout all 5 test cases, Backtracking kept a maximum of 21 nodes in memory at one time. BFS, however, reached a maximum of 371,196 nodes. In that test case, the 8×8 puzzle, there were 21 cages, compared to 20 in the 7×7. This means each cage had more cells, leading to far more combinations and permutations of numbers satisfying the constraints. In fact, the 8×8 puzzle had an average cage size of 3.05, and the 7×7 only had an average of 2.45. It is clear that cage size has a large impact on problem complexity when actions are represented as filled cages.

Good use of figure reference

Figure 2 shows the corresponding runtime with Backtracking on the same set of problems, with a regression yielding $R^2 = 78.43\%$. Since Backtracking fully explores the depth of the tree before the breadth, it fills every cage until it reaches one with no legal numbers, at which point it undoes its previous decisions and moves on to the next possibility. With the depth-first approach, it should run approximately as fast as BFS, with their difference in execution time increasing as puzzles grow in size.

Figure 3 shows a comparison of the two algorithms on the same problem set. As expected, a regression on the two sets of data show that Backtracking is expected to run slightly faster from small problems, with an significant speed increase as problem sizes grow. BFS has a best-fit function

Great use of technical analysis

$$K_{\text{BFS}} = 3 \times 10^{-8} e^{2.882x} \tag{3}$$

Backtracking is fit with the function

$$K_{\text{BT}} = 7 \times 10^{-8} e^{2.607x} \tag{4}$$

Good use of quantification

Following these lines, Backtracking is expected to run approximately 6 times faster than BFS on a 10×10 puzzle. However, due to the small sample size, this difference is within the margin of error, so this result does not indicate any statistically-significant difference in execution time.

Analyzing the space requirements for the two algorithms show a more stark contrast. BFS held a maximum of 371,196 nodes in memory at once, taking exponentially more space with the increasing puzzle size. The best-fit function predicts $1.09 * 10^6$ nodes will be held for solving a 10×10 puzzle, but the correlation is hard to predict with the small sample size. Nevertheless, Backtracking follows only a linear increase of space complexity as puzzle size increases, projecting just 30 nodes for a 10×10 puzzle.

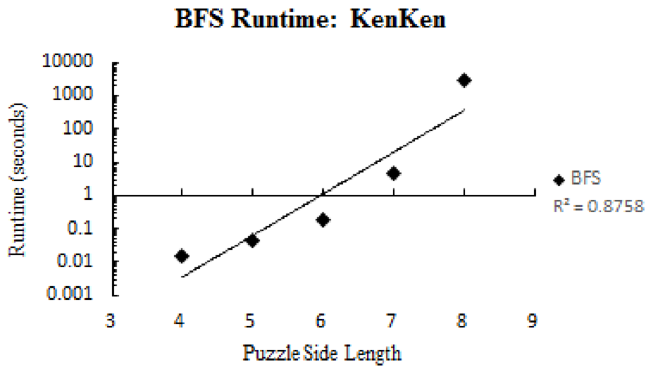


Fig. 1: An exponential regression for the BFS runtime on KenKen, measured in seconds.

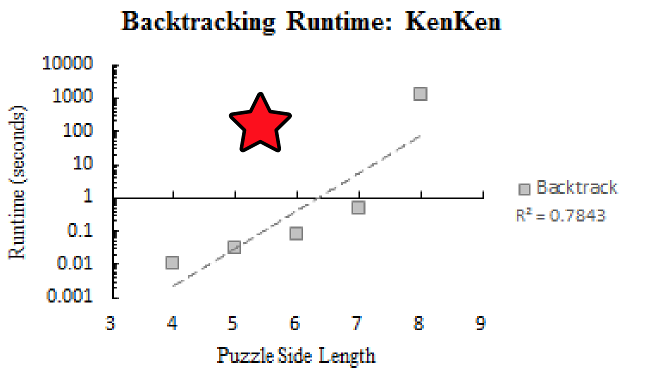


Fig. 2: An exponential regression for the Backtracking runtime on KenKen, measured in seconds.

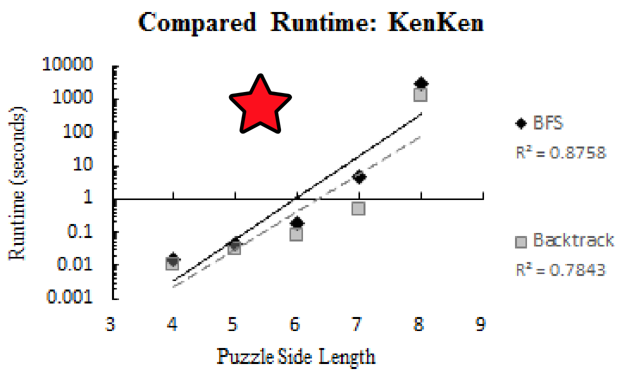


Fig. 3: A comparison of BFS and Backtracking on the same KenKen problems.

Adding that the same problems were used for both may be more information than is needed in a caption. May be better suited in with the explanation of the experiment

B. Cryptarithmetic

The complexity of Cryptarithmetic problems is not as closely tied to domain changes as with KenKen. Since every letter represents a base-10 digit $0 - 9$, the choices for any letter remain constant once 10 letters have been used. Instead, complexity from the problems may grow from the number of characters in each word, or the number of words used in the problem. An analysis of the correlation between execution time and the number of characters per word shows no measurable correlation, narrowing down the source of complexity to the number of words used.

To test the algorithms on Cryptarithmetic, many puzzles of varying lengths were randomly generated using between 4 and 19 words. The increasing number of words leads to large operations, through which carry values must be tracked from all previous operations, slowing down execution and leading to longer runtimes, especially for multiplication problems.

In this Cryptarithmetic implementation, actions correspond to assigning values to every unassigned letter in a column of letters, starting from the right and working leftward. Valid actions are pruned by applying the problem's operation to the columns starting on the right and testing whether the answers correspond to the obfuscated answer in the bottom row. The carry from previous operations is kept in intermediate steps, and it ignored once the operation has been applied to all of the columns up to and including the current action. The calculation time for this is negligable, since it simply uses a list reduce with the operator across the filled columns.

For BFS to solve a Cryptarithmetic problem, it generates every combination of letter-digit pairs before moving to the next column. This leads to its usual large memory requirements, and the wasted branches that it did not need to explore. **Figure 4** shows the expected exponential growth of the search time as the problem complexity grows. One 16-word puzzle took nearly 4 hours to solve, and 18-word puzzles had an average time of 45 minutes.

Figure 5 shows the runtime of Backtracking the same set of problems, and it has a similar exponential increase in execution time to that of BFS. The two follow very similar patterns in their execution times, which stems from the random average difficulties of the 80 problems of each of of 16 sizes.

Performing a regression on the two sets of data shows that BFS actually begins to perform slightly better than Backtracking as the problem sizes increase. The BFS data follows the curve

Great use of technical analysis

$$C_{\text{BFS}} = 0.279e^{0.496x} \tag{5}$$

Backtracking is fit to the function

$$C_{\text{BT}} = 0.170e^{0.535x} \tag{6}$$

These two lines have an intersection point at $x = 12.55$, meaning problems with 12 or fewer words are projected to be solved faster by Backtracking, and larger puzzles are expected to be solved faster by BFS. However, with such small data sets, most BFS runtimes are larger than those of Backtracking, but the lines are offset by a few isolated cases. The small difference between the two lines over the same sets of data is statistically insignificant and within the margin of error, so no definitive conclusions can be made about the execution times.

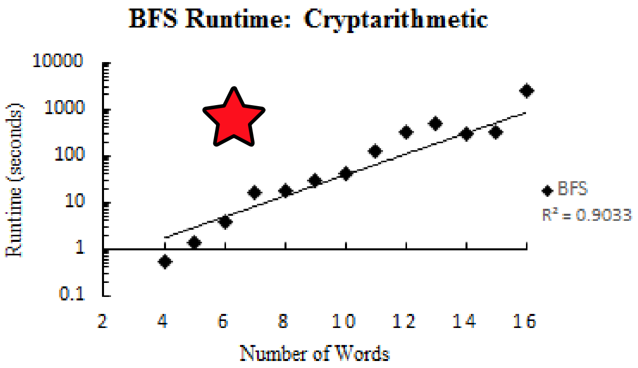


Fig. 4: An exponential regression for the BFS runtime on Cryptarithmetic, measured in seconds.

Graphs are used and organized in a way that is easy to follow

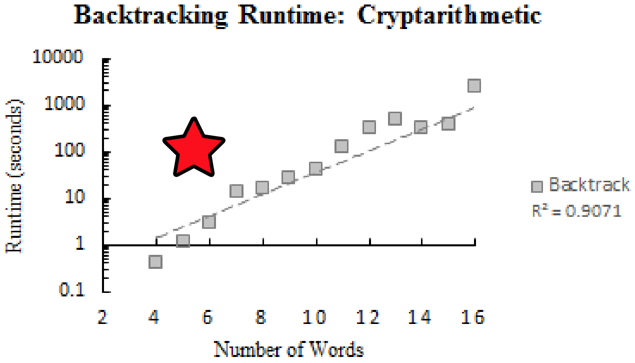


Fig. 5: An exponential regression for the Backtracking runtime on Cryptarithmetic, measured in seconds.

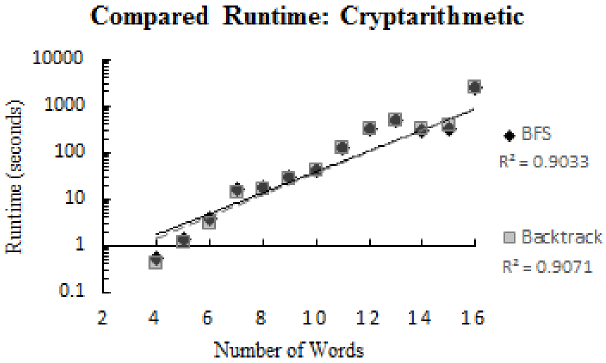


Fig. 6: A comparison of BFS and Backtracking on the same Cryptarithmetic problems.

Similar to KenKen, Backtracking only ever held a maximum of 21 nodes in memory at once, with an average maximum of 6.42 nodes across all 160 test cases. BFS, on the other hand, held a maximum of 371,196 nodes at once, with an average maximum of 109,474.4 across all tests. Since nodes are represented by dictionaries of letter-number pairs, the large number of nodes in memory did not cause any real problems on the computer that was executing the tests, but with larger problems or problems with a larger number of states, this can cause real issues. In addition, problems whose storage need exceed the size of the machine's RAM will need to access addition storage, slowing execution down further.

IV. Conclusion and Future Work

In both KenKen and Cryptarithmetic, Backtracking was found to search solution trees in approximately the same amount of time as BFS, and was much smaller in necessary storage. Cryptarithmetic tests suggested that BFS may begin to run faster than Backtracking as problem sizes increase, but this difference is small and would require further research to prove. In addition, time constraints for storage of the large sets of data that BFS stores must be taken into consideration when extending the algorithm into larger problems. Backtracking stores one state and several actions, which are nearly always smaller than states, while BFS stores every state it has visited. In cases with small state structures, this imact is minimal, but it must also be taken into consideration.

Backtracking has a distinct advantage no matter the puzzle size in KenKen, which derives its complexity and its domain size from the side lengths. On average, Backtracking will only need to search half of the tree before finding a solution, which leads to substantial time savings while problem sizes increase. BFS will store exponentially more data as the puzzles grow, and it will be slowed by not only its computation speed but by its need for extra storage.

In Cryptarithmetic, the advantage of Backtracking over BFS is not as clear, and will require further exploration. With a domain locked at 10 values, increasing problem sizes seem to prove more problematic for Backtracking to solve, and it appears to slow as its wrong guesses lead to larger and larger detours. Because of the pruning techniques involved in Cryptarithmetic, the tree is not a constant depth, and Backtracking may pursue many routes that BFS will not need to investigate. In addition, further exploration is required of the problem difficulty caused by other factors such as operation and word length, and whether varying word lengths have any effect.

Further work will include larger problem sizes for KenKen, as well as more puzzles of the same size to average out individual complexity variations. Cryptarithmetic can be expanded to include problems of different bases, more tests with word lengths, and more total words to see if the experimental prediction holds and BFS does out-perform Backtracking with larger problems.

KenKen tests in the future will require puzzle generation to ensure data sets that are large enough to average out variations in individual puzzle difficulties, leading to more statistically significant conclusions.

References

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proc. Int. Conf. High Performance Comput., Netw., Storage Analysis*, pages 1–11, November 2010.

[2] T. Everitt and M. Hutter. An analytical approach to the bfs vs. dfs algorithm selection problem, 2015.

[3] O. Johanna, S. Lukas, and K. V. I. Saputra. Solving and modeling ken-ken puzzle by using hybrid genetics algorithm, 2012.