Graph Search as a Feature

in Imperative/Procedural Programming Languages

by

Christopher George Henderson

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2018 by the
Graduate Supervisory Committee:

Ajay Bansal, Chair
Timothy Lindquist
Ruben Acuna

ARIZONA STATE UNIVERSITY

May 2018

ABSTRACT

Graph theory is a critical component of computer science and software engineering, with algorithms concerning graph traversal and comprehension powering much of the largest problems in both industry and research. Engineers and researchers often have an accurate view of their target graph, however they struggle to implement a correct, and efficient, search over that graph.

To facilitate rapid, correct, efficient, and intuitive development of graph based solutions we propose a new programming language construct - the search statement. Given a supra-root node, a procedure which determines the children of a given parent node, and optional definitions of the fail-fast acceptance or rejection of a solution, the search statement can conduct a search over any graph or network. Structurally, this statement is modelled after the common switch statement and is put into a largely imperative/procedural context to allow for immediate and intuitive development by most programmers. The Go programming language has been used as a foundation and proof-of-concept of the search statement. A Go compiler is provided which implements this construct.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

DECLARATIVE PROGRAMMING

Declarative programming languages often ask only of the user that they provide a desire, or goal, that they wish to achieve. The operational semantics of that goal are out of the user's hands as the language's runtime environment makes decisions that the user is not a part of.

```
SELECT *
FROM Employee
WHERE name = "Alice"
```

"Retrieve for me all employees whose name is Alice". SQL is perhaps the most well known declarative language on the planet. Exactly how this data is stored or how the system retrieves it is usually of little concern to the user.

That is, declarative languages allow for the easy definition of a goal, however defining *how* that goal is reached is often difficult, if not impossible.

Conversely imperative/procedural programming languages allow for the trivial definition of *how* to achieve a goal. These languages are stemmed from the tradition of Turing Machines and are the predominant languages used in both industry and academia. Outside of database management systems and artificial intelligence research, imperative/procedural languages dominate much of the world (although functional languages have been making a resurgence). For whatever the reason may be (whether it be psychological, machine performance, sheer cultural momentum, or something else altogether) these languages dominate and nearly every new programmer's first language is a member of the imperative/procedural family tree.

However, those who delve into the world of declarative programming often emerge with a great appreciation for the paradigm. They begin to find joy in releasing themselves from the tyranny of manipulating physical memory and relish the transformation from what was a large, and questionable, procedure to a collection of small, and trivially true, facts.

If we are to use imperative/procedural languages, can we not adopt particular features from the declarative paradigm regardless? Afterall, users of imperative/procedural languages rarely argue for the “purity” of the paradigm. Indeed, the Rust programming language is a systems language which implements a classic declarative feature - pattern matching. [1] What else could the more “traditional” languages borrow from the world of declarative programming?

Of the long list list of features commonly found in declarative programming languages, we shall focus our discussion on automatic *backtracking*. Ultimately we shall attempt to port automatic backtracking into an already existing imperative/procedural programming language. Throughout the discussion we will frequently return to the NQueens [2] problem as the canonical example of a backtracking problem.

Backtracking

It is clear that it is *possible* to provide backtracking in a generic, opaque, fashion. Afterall, backtracking is a feature of the Prolog programming language, which Prolog calls a *Proof Search*. [3]

```
f(a).
f(b).

g(a).
g(b).

h(b).

k(X) :- f(X), g(X), h(X).
```

Given k(X), Prolog will succeed in unifying f(a) and g(a) but fail to unify h(a). Prolog will then bactrack and attempt to unify f(b), g(b), and h(b) which all succeed, thus unifying X with b.[3]

What is not clear is whether backtracking can be provided in a more transparent and direct (I.E. imperative) programming language without burdening the user too greatly. In order to succeed at this, we must find a quintessential backtracking algorithm. One which can solve any backtracking problem, yet one whose implementation can be neatly hidden away by a compiler and requires only minimal information from the user. We discovered a candidate algorithm which asserts the following.

In order to apply backtracking to a specific class of problems, one must provide the data *P* for the particular instance of the problem that is to be solved, and six procedural parameters, *root*, *reject*, *accept*, *first*, *next*, and *output*. These procedures should take the instance data *P* as a parameter and should do the following:

1. *root*(*P*): return the partial candidate at the root of the search tree.
2. *reject*(*P*,*c*): return *true* only if the partial candidate *c* is not worth completing.
3. *accept*(*P*,*c*): return *true* if *c* is a solution of *P*, and *false* otherwise.
4. *first*(*P*,*c*): generate the first extension of candidate *c*.
5. *next*(*P*,*s*): generate the next alternative extension of a candidate, after the extension *s*.
6. *output*(*P*,*c*): use the solution *c* of *P*, as appropriate to the application.

The backtracking algorithm reduces the problem to the call *bt*(*root*(*P*)), where *bt* is the following recursive procedure:

```
procedure bt(c)
  if reject(P,c) then return
  if accept(P,c) then output(P,c)
  s ← first(P,c)
  while s ≠ Λ do
    bt(s)
    s ← next(P,s)
```

A candidate backtracking engine and interface. [4]

This is a promising candidate. However, it is an *assertion* and is not trivially true, so first we must explore the validity of this claim.

Discovery

Our assertion is that, given the six procedures root, reject, accept, first, next, and output that any backtracking algorithm can be conducted. This is not easy to show outright, so it is useful to first construct working examples so that the behavior can be observed. We shall accomplish this by using higher order functions written using Python. We shall also use the NQueens problem as the canonical example now, and throughout this paper.

```python
def backtrack(P, root, accept, reject, first, next, output):
    if reject(P, root):
        return
    P.append(root)
    if accept(P):
        output(P)
        P.pop()
        return
    s = first(root)
    while s is not None:
        backtrack(P, s, accept, reject, first, next, output)
        s = next(s)
    P.pop()
    s = next(root)
    if s is not None:
        backtrack(P, s, accept, reject, first, next, output)
```

A higher order procedure implementing a backtracking algorithm.

Immediately we see small differences between the provided pseudo code and the implementation. Chief among them is that the `P` data structure, representing the solution thus far, is being managed by the algorithm itself rather than the userland code (the assumption made before was that the user managed this structure themselves). Additionally, root is not a procedure anymore, rather it is a given node that is recursively taken to be the root of a subgraph.

Finally, there is an additional call to the next procedure before the backtracking algorithm terminates. This is an artifact of our canonical example being the NQueens problem and that the traditional root for the NQueens problem is the most top-left corner of the board, (1, 1). However, the root (1, 1) itself has siblings. Namely, (1, 2), (1, 3), (1, ...), and so on down the first column. This makes the NQueens problem seem more like a forest rather than a directed acyclic graph. This issue will be addressed further in the *First Choice Generator* section.

Given our higher order procedure, the following is an encoding of the NQueens of 8 problem.

```
N = 8

def reject(P, candidate):
  C, R = candidate
  for column, row in P:
    if (R == row or
       C == column or
       R + C == row + column or
       R - C == row - column):
      return True
  return False


def accept(P):
  return len(P) == N


def first(root):
  # Generate the first extension of the root.
  # That is, if we have a placed queen at 1, 1 (column, row)
  # then this should return 2, 1.
  return None if root[0] >= N else (root[0] + 1, 1)


def next(child):
  # Generate the next alternative extension of a candidate,
  # after the extension s.
  # So if first generated (2, 1) then this should produce (2, 2).
  return None if child[1] >= N else (child[0], child[1] + 1)


def output(P):
  print(P)

backtrack([], (1, 1), accept, reject, first, next, output)
```
Procedure definitions for the NQueens problem with a call to the backtracking engine.

Dynamic Description of a Graph and Separation of Concerns

We must reflect on the semantics of these facts and procedures at a signature level. That is, not the semantics of the body, which describes a given graph, rather we must understand what the first, next, accept, and reject procedures are for *all* graphs. They are, in fact, dynamic *descriptions* of a graph.

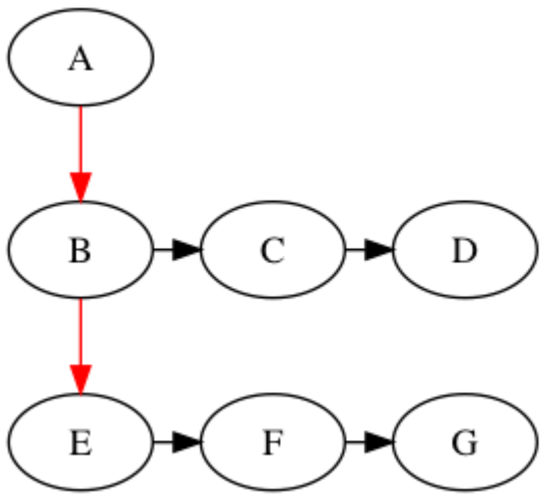

Figure 1: A is a root which produces B via first. C and D are produced via next. B is a root which produces E via first. F and G are produced via next. And so on recursively.

first and next are responsible for the dynamic generation of a graph. accept and reject are consultants on the current state of the solution. Given the ability to request any set of nodes in a graph, and the ability to consult on the state of a solution, a caller can select to traverse that graph in any fashion that they desire.

This accomplishes a profound separation of concerns that will serve as the foundation for the search statement. We shall refer to the caller as the “engine”. We shall refer to the callee as the “userland”.

The engine need not ever concern itself with the semantics of the graph. To the engine a node-is-a-node and as long as it is receiving nodes and consultations it can continue its algorithmic work.

The userland need not ever concern itself with the algorithm of the traversal. To the userland, all that must be done is to provide nodes when requested and consult when the engine believes it may have come across a solution.

This uncoupling of the graph algorithm and the graph semantics is key to providing a first-class interface embedded within a language. That is, the algorithm and the semantics are no longer entangled. Rather they *communicate* with each other.

```
let procedure backtrack():
    while algorithm:
        semantics
        if semantics:
            algorithm
        algorithm
```

```
Algorithm → Semantics
            Semantics
Algorithm ← Semantics
Algorithm
Algorithm → Semantics
```

Left: The algorithm and the semantics intertwined with each other. Right: The algorithm and the semantics communicate with each other, however they never violate each other.

Similar to how a C style for construct separates the exact implementation of a loop from the user's desired loop behavior, we can separate the exact implementation of a graph traversal from the user's desired graph.

REFINEMENT

Six mandatory pieces of information and procedures from the user is, however, burdensome and the resulting construct would appear, at best, baroque. In order to make the construct more clear, concise, and easier to pronounce we shall attempt to identify reductions in tahese requirements. Namely, we will collapse the first and next procedures into the singular children procedure, we will make accept and reject optional, and we will eliminate output altogether. Ultimately, we will identify that we have discovered not only a backtracking engine, but in fact a much more general *searching* engine.

Output

We shall eliminate the output procedure altogether. It is unnecessary as any desired “output” behavior can just as easily be placed within the accept procedure before returning true.

```
let procedure backtrack(root, solution):
        if reject(root, solution):
                return
        if accept(solution):
                output(solution)
                return
        for child ∈ children(root):
                backtrack(child, solution)
```

```
let procedure backtrack(root, solution):
        if reject(root, solution):
                return
        if accept(solution):
                return
        for child ∈ children(root):
                backtrack(child, solution)
```

Left: Output as called by the engine. Right: Output is left to the accept procedure to call, if called at all.

```
let procedure accept(solution):
        if valid(solution):
                output(solution)
                return true
        return false
```

An example accept procedure which calls output before returning true.

The Lineage Problem

first and next work well for problems where siblings can easily identify each other. Problems such as the NQueens can be expressed as the appropriate increments of the given node’s current state.

```
let procedure first(queen):
        return (queen[0] + 1, 1)

let procedure next(queen):
        if queen[1] + 1 > N:
                return nil
        return (queen[0], queen[1] + 1)
```

The NQueens first and next procedures readily generating each other.

This works well in code, but is rather unwieldy in representation. A graphical representation of the NQueens problem, solved using first and next, is what most would consider extremely awkward.

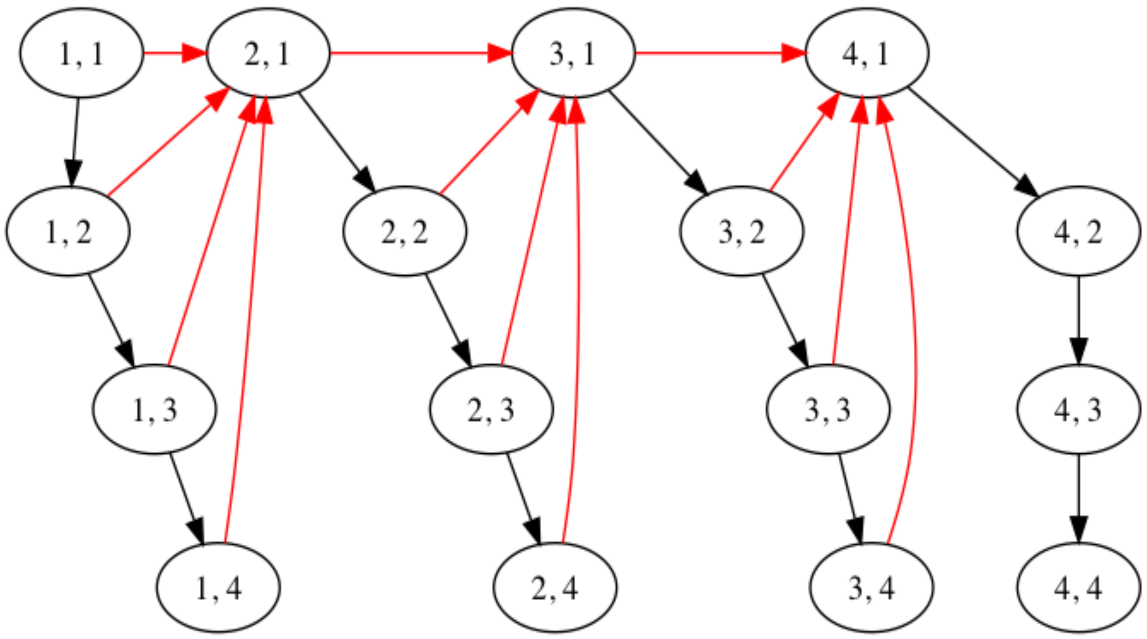

Figure 2: The NQueens visualized with first and next procedures. Calls to first are in red. Calls to next are in black.

Every node produces the first row of the next column as its first, and every node produces the next row of its own column as its next. What worked so well in code is far from what a programmer would naturally think of as a natural and reasonable DAG.

However, being unnatural is the least of the concerns for the first and next constructs. *Lineage* becomes a very real, non-stylistic, concern for the algorithm. In a problem which has lineage, the answer to one's next procedure is dependent upon the ancestry of the given node - any change in your ancestry changes who your siblings are.

Lineage is not an issue for the NQueens problem. For example, the node at (2, 1) does not require any specific knowledge of its ancestry in order to produce its siblings - they will always be [(2, 2), (2, 3), (2, 4)]. Changing the parent of (2,1) to from, say, (1, 1) to (1, 2) will not change this fact.

However, what about the maze problem? For any given node, its first can be any arbitrary selection of the four cardinal directions. However, once that first is visited, who are its siblings?

```
0, 0, 0, 0, 0, 0
1, 1, 0, 0, 0, 0
0, 1, 0, 0, 0, 0
0, 1, 1, 1, 1, 1
0, 1, 0, 1, 0, 0
0, 0, 0, 1, 0, 0
```

Figure 3: Who are the siblings of the red node? Depending on where we came from, they can either be the blue nodes [(3, 2), (3, 4), (4, 3)] (came from the right) or the orange nodes [(2, 2), (2, 4), (1, 3)] (came from the left). We can only know for certain if we know who the parent was. And even then, it is not simple to generate siblings without a cycle occurring.

All of the possible siblings of a given node are all of the children of that nodes' possible parents. That is, in order to ascertain your siblings, you must at minimum have a notion who your parent was. However, that is still not enough. If the childrens' only knowledge is who their parent is, then the possibility of an infinite loop occurs as siblings begin producing each other with no notion of who has already been visited. Two immediate solutions to this infinite loop presented themselves.

1). Introduce a visited set that the siblings pass around to each other as they are visited.
2). Introduce an unvisited set that the parent produces and passes off to its first child. The siblings then proceed to remove themselves from the set as they are discovered.

These solutions introduce cycle detection in the small. Cycle detection in the large (I.E. for the maze problem as a whole) must still be present. Within using the first/next, either of these approaches were candidates for becoming idioms for solving similar problems. The second, unvisited, solution was what we elected to use when first solving such problems that require a lineage.

This solved the issue at the moment, however it left us unsatisfied. Suddenly the user is no longer solving their own graph search. Rather they are implementing awkward micro-level cycle detection that would otherwise never have existed in their original problem if they had just written their solution from scratch. That is, the first/next construct has made their program significantly *harder to write*.

Children

The user provided procedures *describes* the graph while the underlying engine *solves* the graph in a black-box manner. That is, the root procedure describes where to start, accept and reject describe where to stop, and first and next describe how to *walk* the graph.

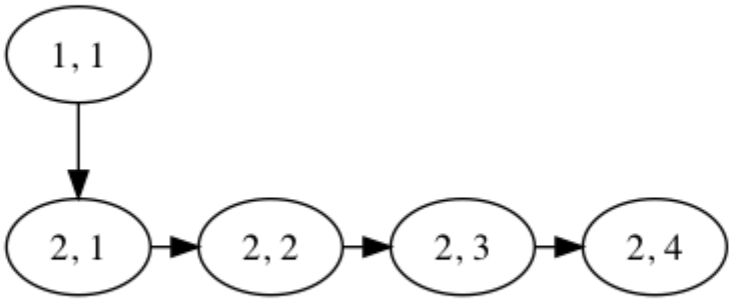

Figure 4: NQueens (1, 1) node generating (2, 1) via first, with the rest of the second column being generated via next.

We have identified that using first/next to walk a graph is, at best, conceptually awkward and at worst difficult in implementation. However, the key observation is not in *how* first/next are describing a graph, but rather that it *is* describing a graph. As such, is there a better, more natural, way to describe any graph?

Parent-child relationships are the classical way of thinking about directed graphs. The concept can even be used ephemerally on non-directed graphs (the current node being visited could be thought of as being the parent, with any other nodes connected by vertices being children - the moment any of the children are visited, the visited child becomes the parent and the parent becomes a child).

```
let V be the set of all vertices
let E be the set of all edges
let G = {V, E}
let procedure Connected(Edge, Parent, Child) ->
  true ↔ Edge = (Parent, Child)
let procedure Children(G, Parent) ->
  {Child: Child ∈ V ∧ ∃Edge ∈ G.E Connected(Edge, Parent, Child)}
let procedure Walk(G, Root) ->
  for each Child ∈ Children(G, Root)
    walk(G, Child)
```

Psuedo code of a the walking of a DAG.

The above is a classical Depth-First Search, sans cycle detection. Since cycles are not necessarily present for any and all graphs, cycle detection is left to the user to define as desired.

```
procedure DFS(G,v):
   label v as discovered
   for all edges from v to w in G.adjacentEdges(v) do
       if vertex w is not labeled as discovered then
           recursively call DFS(G,w)
```

Psuedo code of a Depth-First Search. [5]

Swapping out first/next for children has not only made our representations more natural, but it is has made the engine *verifiable*.

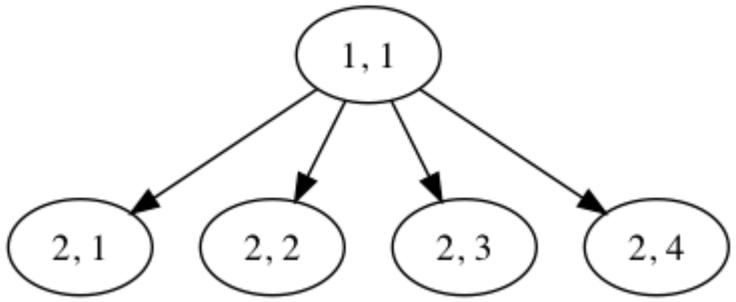

Figure 5: NQueens (1, 1) node generating the entire second column as children.

It is trivially true that, given a procedure that can return a list of children of a given node, an engine can perform a walk of that graph in any fashion that it desires.

```
let procedure DFS(parent):
      ….
      for child ∈ children(parent):
            DFS(child)
      …

let procedure BFS(parent):
      while len(queue) is not 0:
            ….
            parent = get(queue)
            for child ∈ children(parent):
                  put(queue, child)
            ...
```
Psuedo code of a DFS and BFS with no cycle detection.

From hereon, we will abandon the use of the first and next procedures in favor of the children procedure. Our engine is now:

```
let procedure backtrack(root, solution):
      if reject(root, solution):
            return
      if accept(solution):
            return
      for child ∈ children(root):
            backtrack(child, solution)

backtrack(root(G), ())
```
Psuedo code of the backtracking engine with the children procedure replace the first and next procedures.

We are down from six mandatory procedures to five.

A Tree Becomes a Forest

The move to the children procedure has an interesting affect on how the graph search is initialized. Previously, with the first and next procedures, the user provided root was responsible for generating its siblings.

Graphs which have only one possible first choice look, relatively speaking, natural.

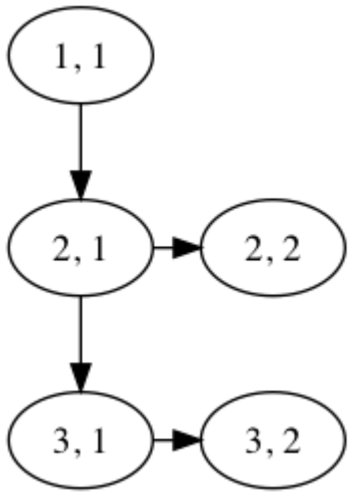

Figure 6: The only possible first choice to this graph is the node (1, 1)

There is only one possible starting point for the graph search, and therefore only one possible root. However, for searches that have multiple possible entry points, the graph becomes trapezoidal.

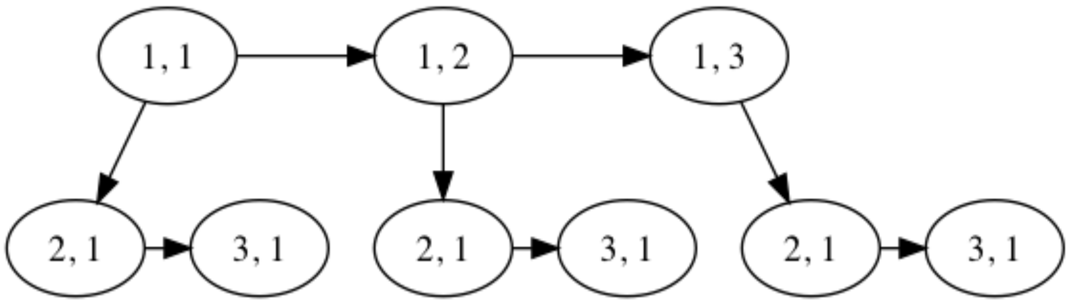

Figure 7: In the NQueens problem, each first choice of a queen placement generates the next first choice as a sibling.

This is strange, but workable. However, with the move to the `children` procedure, the above graphs transforms from a trapezoidal DAG into a *forest*.

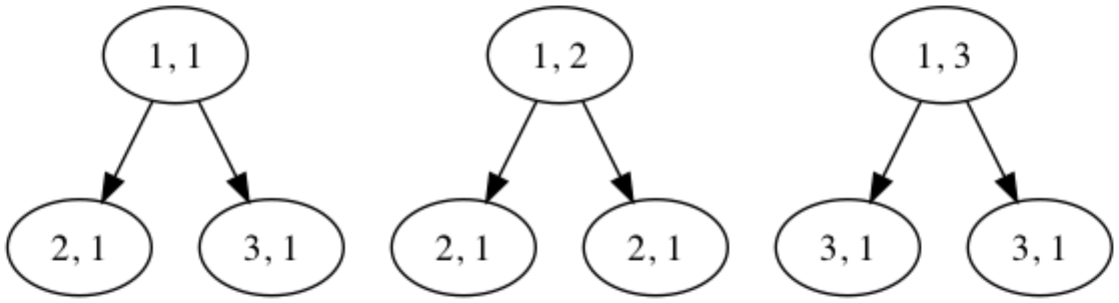

Figure 8: A graph search with multiple entry points would have to be executed multiple times as this is actually a forest.

The nodes at level zero can no longer generate each other. As such, for the NQueens problem, a search for a solution that starts with a queen at (1, 1) is separate from a search for a solution that starts with a queen at (1, 2).

This problem presents itself frequently. When coloring a map, which U.S. state do we start from? When solving a maze, there may be multiple entries, which entry do we start from?

It is certainly possible to run this search in a loop, where each iteration of the loop generates the next root node to search from. This solution, however, was unsatisfactory. These problems are rarely thought of as being forests, so it is inappropriate to treat them as such. What we need instead is a node which represents *the point from which no choice has yet been made*.

The First Choice Generator

Every graph has, external to the universe of the graph itself, a node which selects all of the possible first choices of that graph. It is the node from which no choices have yet been made. We will call this node the *First Choice Generator* (FCG). The FCG is not a member of the graph itself nor will it ever be a member of the solution to a particular search. The FCG acts solely as an oracle which informs the algorithm of all possible starting points for a given search. The definition of the FCG and its graphs is the following recurrence relationship:

1. A FCG is followed by zero or more graphs.
2. A graph is a node followed by zero or more graphs.

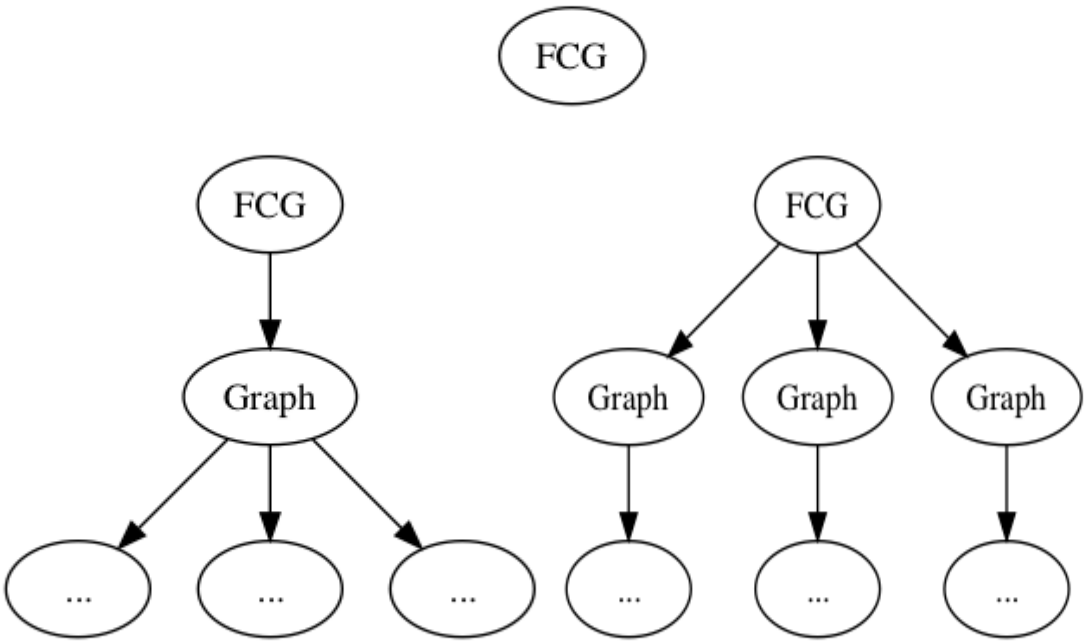

Figure 9: An FCG followed by no graph. An FCG followed by a single graph. An FCG followed by multiple graphs.

Using an FCG, we can connect the supposed NQueens forest into a single tree.

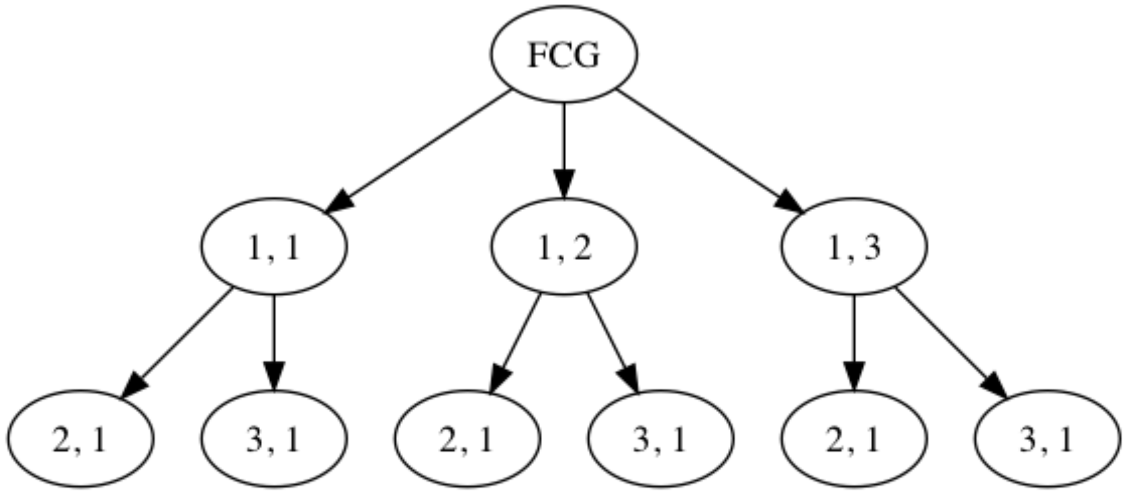

Figure 10: An FCG connecting all first choices into a single DAG.

All parents generate their children, and all possible first placements of a queen are connected into a single tree using an FCG. At last, the NQueens problem looks *natural*.

FCG Replaces Root

The use of an FCG obviates the presence of the root procedure. The FCG has as its children all possible starting points of the graph search. That is, rather than provide a procedure which accepts a graph and returns a root, the FCG can be given as a naturally occuring parent node to the children procedure.

If the children procedure for the NQueens problem is defined as follows:

```
N = 8
let procedure children(parent):
      let l list
      let column = parent[0] + 1
      for row 1...N:
            append(L, (column, row))
      return l
```

Example children procedure accepting an FCG and producing all first choices for a graph search.

Then all first choices of the NQueens problem can be generated by the call children((0, 0)).

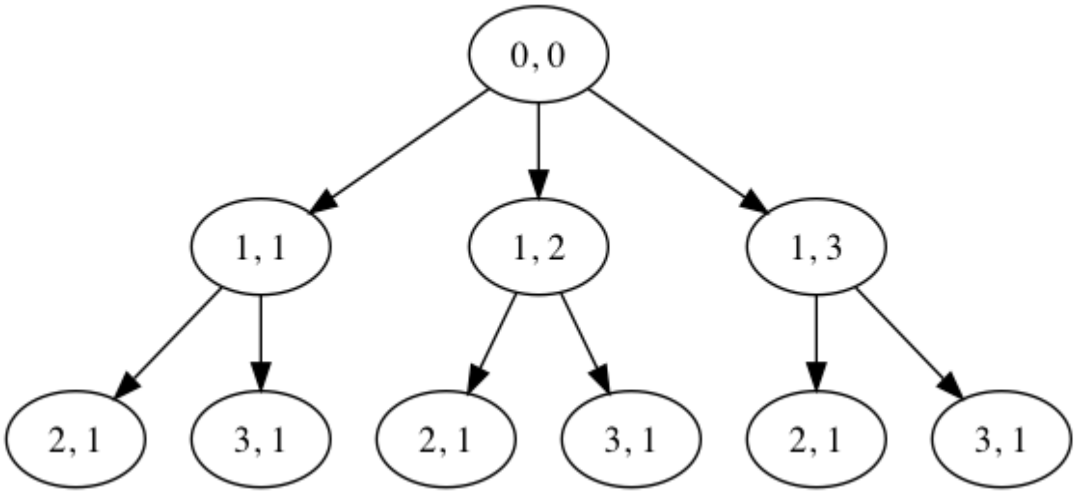

Figure 11: The node (0, 0) is off of the chess board, but it acts as the FCG for the search.

Our definition of the backtrack procedure remains the same. What changes is our *call* to the procedure.

```
let procedure backtrack(root, solution):
      if reject(root, solution):
            return
      if accept(solution):
            return
      for child ∈ children(root):
            backtrack(child, solution)

backtrack((0, 0), ())
```

A call to the backtracking engine providing an FCG in lieu of a root node.

We are down from four mandatory procedures to three.

The Move to a Searching Engine

Up to this point we have been discussing a subset of graph search algorithms - backtracking. We have given mention to several different, or more broad, class of algorithms such as DFS and BFS, however backtracking has remained our focus thus far. Let us discuss how we might generalize our previous discoveries into a more universal *searching* engine.

During the course of the research, we came across several example problems whose accept and reject procedures were either empty (hardcoded to return false) or were completely arbitrary in their presence (E.G. setting a maximum depth or width, just to ensure that the program halted in a timely fashion). This led us to consider *why* these problems appeared to be ambivalent to the accept and reject procedures and whether the results had consequences for our engine. One of the most demonstrative examples was a simple webcrawler.

Constructing a Webcrawler

Let us consider a simple webcrawler. The world wide web is a colossal, directed, often cyclic graph of named nodes - a network. The network is alive, meaning that it is growing or, less commonly, shrinking even as you are reading this. Our little webcrawler would like to capture a snapshot of at least some small portion of the web as it currently is.

Traditional webcrawler technology describes a seed page, an extraction routine, a frontier of candidate pages, and a database to store results in. Although you will likely select a select a seed page that is conducive to mining the topic you are interested in (E.G. Olympic curling), any web page will do. The extraction routine is any procedure that can download, parse, and extract outgoing links from the page. The frontier is any data structure that can aggregate candidate links, what exactly this data structure *is* is dependent on what type of search you wish to conduct. The database is irrelevant to this discussion. Although irrelevant to this discussion, you will likely also want some sort of cycle detection (pages often have a banner that link back to itself).

Let us decide that our webcrawler will conduct itself in a DFS manner, and thus our frontier must be a stack. We will implement this stack implicitly using function calls recursively.

```
let procedure extraction(seed, database):
      content = download(seed)
      put(database, content)
      links = parse_hrefs(content)
      return links

let procedure crawl(seed, database):
      for link ∈ extraction(seed, database):
            crawl(link, database)
```

A simple webcrawler. Procedures that can write to a database and parse all HREFs from an HTML document are assumed to be available.

Considering the depth, breadth, and growth rate of the web the crawl procedure is unlikely to terminate on its own. Indeed, server maintenance, power outages, and stack overflows are far more likely culprits. Effectively, we can say that this algorithm does not halt.

But what if we did want it to halt? One solution is to introduce a maximum search depth:

```
let depth = 0
let max_depth = 1000

let procedure extraction(seed, database):
      content = download(seed)
      put(database, content)
      links = parse_hrefs(content)
      depth++
      return links

let procedure crawl(seed, database):
      if depth >= max_depth:
            return
      for link ∈ extraction(seed, database):
            crawl(link, database)
```

A webcrawler limited to a maximum depth and width.

Another way to limit the search is to set a domain restriction. Crawling a single domain is a much more tractable problem than crawling the entirety of the web, so it at least has a chance of halting.

```
let root_domain = us.olympiccurling.org

let procedure crawl(seed, database):
      if domain_of(seed) != root_domain:
            return
      for link ∈ extraction(seed, database):
        crawl(link, database)
```

A webcrawler with a domain restriction. It shall never leave the seed's host domain.

We can combine these two strategies to construct a webcrawler that will only search to a maximum depth on a single domain.

```
let procedure crawl(seed, database):
    if domain_of(seed) != root_domain:
        return
    if depth >= max_depth:
        return
    for link ∈ extraction(seed, database):
        crawl(link, database)
```

A webcrawler limited to the seed's host domain as well as to a maximum depth and width.

The non-halting crawl procedure is simply a renamed DFS.

```
let procedure crawl(seed, database):
    for link ∈ extraction(seed, database):
        crawl(link, database)
```

```
let procedure DFS(root, solution):
    for child ∈ children(root):
        DFS(child, solution)
```

A webcrawler which never halts is merely a DFS that stores its result in a database.

The halting crawl procedure is a type of backtracking DFS where we accept a solution if we have reached maximum depth, and we reject if we are attempting to leave the root domain.

```
let procedure crawl(seed, database):
  if domain_of(seed) != root_domain:
    return
  if depth >= max_depth:
    return
  for link ∈ extraction(seed, database):
    crawl(link, database)
```

```
let procedure backtrack(root, solution):
  if reject(root, solution):
    return
  if accept(solution):
    return
  for child ∈ children(root):
    backtrack(child, solution)
```

A domain and depth limited webcrawler is very similar to the outline of the backtracking engine.

Either, both, or neither of the crawler's reject and accept branches may be present. They are informative of desired stopping points, but they are not required for the webcrawler to function. And, in fact, with neither reject nor accept present the webcrawler becomes a simple DFS over the web, starting from a seed page.

The removal of the reject and accept procedures from our backtracking engine produces the same result - the transition from a backtracking algorithm to a pure DFS.

`let procedure backtrack(root, solution):`
`    for child ∈ backtrack(root):`
`        backtrack(child, solution)`
	`let procedure DFS(root, solution):`
	`    for child ∈ children(root):`
	`        DFS(child, solution)`

Side-by-side of a backtracking engine with no accept nor reject, and a DFS procedure.

Acceptance and Rejection

This spawns the realization that the reject and accept procedures are *optional*. When neither are present, the engine becomes a traditional DFS. When both are present, the engine becomes a traditional backtracking algorithm. When either/or are present, the engine becomes a backtracking algorithm that will never accept/reject.

Search Engine	**Reject**	**No reject**

Accept	`let procedure backtrack(root, solution):`
	`  if reject(root, solution):`
	`    return`
	`  if accept(solution):`
	`    return`
	`  for child ∈ children(root):`
	`    backtrack(child, solution)`
		`let procedure backtrack(root, solution):`
		`  if accept(solution):`
		`    return`
		`  for child ∈ children(root):`
		`    backtrack(child, solution)`

No Accept	`let procedure backtrack(root, solution):`
	`  if reject(root, solution):`
	`    return`
	`  for child ∈ children(root):`
	`    backtrack(child, solution)`
		`let procedure backtrack(root, solution):`
		`  for child ∈ children(root):`
		`    backtrack(child, solution)`

Different implementations of a webcrawler based on different halting criteria.

Applied to our webcrawler, the semantics of the above table are:

Webcrawler	**Reject**	**No Reject**
Accept	`Search through only the root domain, only up to a certain maximum depth.`	`Search only up to a certain maximum depth.`
No Accept	`Search through only the root domain.`	`Search the web.`

Different semantics of a webcrawler based on different halting criteria.

All of these semantics are correct. Therefore the reject and accept procedures are optional.

Generalization

This generalizes our construct, allowing for the à la carte implicit selection of a generic DFS or one of three different types of backtracking. This changes the engine in two ways - the renaming of the engine from backtrack to search and the removal of the mandatory requirement of a reject and accept procedure.

The engine is now as follows. Optional blocks are notated in brackets.

```
let procedure search(root, solution):
      [if reject(root, solution):
            return]
      [if accept(solution):
            return]
      for child ∈ children(root):
            search(child, solution)
```

Psuedo code of the search engine. Blocks encased in brackets are optional.

We are down from three mandatory procedures to one.

CONSTRUCTION

We have thus far shown that a generalized search algorithm can indeed be compressed to a small, ergonomic, signature - provided an FCG and a procedure that, given a node, can return all children of that node, a graph search can be conducted in a DFS manner. Given additional procedures that define a success state and a failure state, the search can be pared back to a backtracking algorithm. Many of our examples and demonstrations thus far have been accomplished through either psuedo-code or the use of a higher order function implemented in Python. Our challenge now is in the *implementation* of this feature, embedded within a imperative/procedural programming language.

Up until now many implementation details have been ignored in favor of talking about general graph definitions and recurrence relationships. This does not mean that the proposed engine is not without technical nuance. This chapter promises to bring these topics to light. This includes construction and management of the solution data structure, stack management, userland isolation, concurrency, type information, and so on.

Our goal is to demonstrate our keyword in a fully functional, and feature rich, programming language, primarily of the imperative/procedural tradition. While we could have constructed a new language built around our core concept, however time and scope simply did not allow for it. As such, we instead chose to embed the new feature in an existing programming language. The target language was initially Python, but quickly shifted to Go.

Design

Shape and Signature

Our goal is to inject this construct into a procedural/imperative programming language in such a way that it flows naturally, and well, with its host language. Factors to that goal are the general aesthetics, relatability, and consistency that the construct conveys to users of the language. Any difficulty in learning the construct *must* stem from understanding the underlying problem and *not* from confusion, or forgetfulness, caused by the syntax.

As previously stated, the two pieces of information required to conduct even the most bare minimum search are a first choice generator and a children procedure. The importance of these directives led to us to investigate their use as the search signature.

Backtrack	DFS
`search` `FCG; children:`	**`search`** `FCG; children:`
`accept:`	`{empty}`
…	
`reject:`	
…	

A backtrack and a DFS.

That is:

```
search <expression>; <procedure_name>:
      [<procedure_name>]:
            <body>
      [<procedure_name>]:
            <body>
```

Search statement grammar.

.

This appears good until we begin to think on how to associate a <body> with the children procedure. Assuming that we are within a language with first class functions we can simply provide a name to a target procedure which as already been declared within scope.

```
let procedure NQueensChildren(parent):
      …

search Queen{0, 0}; NQueensChildren:
      accept:
            …
      reject:
            …
```

Search statement accepting first class functions as a signature paramter.

This, however, quickly becomes burdensome as the user is required to frequently declare closures or to declare the procedure elsewhere in their code. This defeats the desire for immediacy as well as the lack of ceremony that the research is attempting to reach. It may be possible, however, to declare the procedure body within the search signature as a lambda.

```
search Queen{0, 0}; procedure(parent):
                         {…}
                       :
      accept:
            …
      reject:
            …
```

Children procedure defined in the signature as a limbda.

This solution, however, introductions three different levels of indentation in a straddled, uncommon, and very unnatural fashion.

```
        ....
            ....
          ....
```

The three levels of indentation.

It does, however, provide us with our desired immediacy. That is, it satisfies our desire to simply state the procedure as locally as possible. We find that if we simply place the children procedure with its peers that immediacy is maintained while simultaneously correcting the indentation issue.

Backtrack	DFS

`search FCG:`
`      children:`
`            …`
`      accept:`
`            …`
`      reject:`
`            …`
	`search FCG:`
	`      children:`
	`            …`

A backtrack and DFS example with the children procedure moved to the body of the search statement.

Implicit vs Explicit Receivers

In object-oriented programming, the issue of an implicit reciever versus explicit receiver is the discussion of whether or not the programmer must explicitly declare, and name, the instance bound to a given method, or if the instance magically appears within the method's scope at some predefined, and unchangeable, name.

Python	Go

`class person(object):`
`  …`
 	
`  def walk(self, distance):`
`    …`
	`type Person struct {}`
	
	`func (p *Person) walk(distance int) {`
	`    ...`
	`}`

Python and Go using explicit receivers.

In Python, the first formal parameter must always be the instance bound to that method. Naming the receiver self is merely convention, it can be named anything. Similarly, in Go, all methods are prefaced with the type and name of instance bound to that method. Again, naming the receiver as the first letter of the type is simply convention and nothing more.

Java

```
public class Person {
    …
    void walk(int distance) {
        …
    }
}
```

C++

```
class Person {
    …
    void walk(int distance) {
        …
    }
}
```

Java and C++ using implicit receivers.

In both Java and in C++ the individual programmer does not name the receiver to the method. Rather they implicitly receive within their local namespace the object named this. The name of which cannot be changed.

The decision between the two is one borne of style, philosophy, and frankly personal preference and it is one that we must make for our search construct.

Explicit

```
search FCG:
      children(parent):
            …
      accept(solution):
            …
      reject(candidate, solution):
            …
```

Implicit

```
search FCG:
      children:
            …
      accept:
            …
      reject:
            …
```

Comparison of explicit and implicit receivers in the search statement.

In the case of the explicit receiver the programmer is responsible for naming each parameter of each procedure. The benefit to this approach is that the programmer is explicitly aware of the objects available to them within their scope. Additionally, the programmer may chose to name the parameters to something more descriptive to their business logic. The construct, however, is verbose and perhaps burdensome.

In the case of the implicit receiver the programmer receives magical object references within each procedure. This has the benefit of making the code appear cleaner, more organized, and consistent. All of that at the cost of having “magic” in the code which requires the programmer to simply be in the know that they are receiving these object references.

While our research attempted to maintain the general style and flow of our candidate language, it was not our goal to carry the torch of that language. That is, despite whatever the candidate language’s opinion is on the matter, our decision was to be irrespective of that opinion.

As such, we elected for the use of *implicit receivers*. Specifically, the parameters shall always be named node and solution.

The name solution was chosen merely at a whim, as it is both descriptive and intuitive, albeit relatively long for a mandatory name. The name node, however, required forethought. Two of our procedures refer to some object within their bodies that is representative of a node in the graph. In one context, the object is semantically the parent node being visited. In the other, the object is a candidate node that may, or may not, be rejected. From the outset, we named these two parameters parent and candidate in their respective blocks. However, remembering two different, mandatory, names became burdensome, even on the very creators of the construct itself. We shifted to the shared name, node, upon reflection of the common name, this. The use of the name this in Java and C++ is accepting that it is non-descriptive of the business logic, however it is very descriptive of what it is in the context of the language - it is the self-referential object. node is similar in that it does not describe the context of the object in the business logic, but instead recognizes that the object is just some node within the graph.

This discussion on implicit versus explicit receivers, as well as their names, is largely stylistic, philosophical, and psychological. The decisions made in this section have no technical bearing on the research itself and we provide no proof nor study on the effects of these decisions.

Strong Closures vs OOP

The venerable master Qc Na was walking with his student, Anton. Hoping to prompt the master into a discussion, Anton said "Master, I have heard that objects are a very good thing - is this true?" Qc Na looked pityingly at his student and replied, "Foolish pupil - objects are merely a poor man's closures."

Chastised, Anton took his leave from his master and returned to his cell, intent on studying closures. He carefully read the entire "Lambda: The Ultimate..." series of papers and its cousins, and implemented a small Scheme interpreter with a closure-based object system. He learned much, and looked forward to informing his master of his progress.

On his next walk with Qc Na, Anton attempted to impress his master by saying "Master, I have diligently studied the matter, and now understand that objects are truly a poor man's closures." Qc Na responded by hitting Anton with his stick, saying "When will you learn? Closures are a poor man's object." At that moment, Anton became enlightened. [6]

A stated goal of the research is to insert this feature into a primarily imperative/procedural programming paradigm. The reason for this is to showcase it as a first class feature of the language, and not merely a linear combination of extant features. We also wanted the ability to say immediately, and with little ceremony, exactly what one wanted to say about their target graph.

For theses reason OOP languages, in particular Java, were avoided. The proclivity to use the interface and class keywords to solve all problems would have led to something akin to the following.

```
public abstract class Backtracker {
	...
}
public class NQueens extends Backtracker {
	...
}
ArrayList<ArrayList<Queen>> solutionList = new ArrayList<ArrayList<Queen>>();
new NQueens(8, solutionList, new Queen(0, 0)).backtrack();
```

A skeleton of a Bactracker class and implementation in an object oriented fashion.

In fact, from outset of the research when Java was being considered as a target language, the PoC code looked very similar to the above example.

While this is not unattractive, it is hardly ergonomic. A similar construct that faces this problem is the common for...each style of looping. In many older, or OOP, languages this is solved by an iterator pattern.

```
Iter iter = container.iter();
while (iter.HasNext()) {
	...
}
```

An iterator pattern.

A perfectly reasonable solution. However most modern languages choose to make an iterator a core feature of the language.

Python	`for index, obj in enumerate(container):`
	 `    ...`

Go	`for index, obj := range container {`
	 `    ...`

	`}`

Rust	`for (index, obj) in container.enumerate() {`
	`    ...`
	`}`

The iterator pattern implemented by the language in Python, Go, and Rust.

A built in iterator pattern may not afford the user any additional power over the iterator pattern, however it is compact, intuitive to say, and consistent throughout the entire use of the language (I.E. not in peril of individual developers slightly modifying the contract due to personal taste). These are the traits that we desired in the search construct.

However, there *is* state involved. Or rather, a *context* is involved with each graph search. OOP is well known, indeed *designed* for, handling state well. Let's look again at the construction of the NQueens problem in an OOP fashion.

```
new NQueens(8, solutionList, new Queen(0, 0)).backtrack();
```

A call to the object oriented NQueens backtrack implementation.

The first argument to the constructor of the NQueens class is the size of the problem, in this case eight. The second argument is a resizable array. Presumably, after execution of bactrack, this list will contain all computed solutions to the NQueens problem. This collection could not be given to the backtrack method because the base class does not allow for that particular signature. Alternatively, the NQueens class could assume the desire for this list of solutions and allocate the list within its own constructor. Or the NQueens class could offer a constructor that does not aggregate solutions at all, choosing instead to visit each solution and throw it away.

NQueens as OOP	Given	Constructed
Aggregate	`new NQueens(8, solutionList, new Queen(0, 0)).backtrack();`	`new NQueens(8, true, new Queen(0, 0)).backtrack();`
Do Not	`new NQueens(8, null, new Queen(0, 0)).backtrack();`	`new NQueens(8, false, new Queen(0, 0)).backtrack();`

Aggregate		

Different ways to construct and call the NQueens problem in an object oriented fashion.

Already, quite a bit of decision making and style enforcement are occurring just to answer the question of how to persist the solutions to the question. One could imagine further design patterns that may aid in this problem.

Now let us consider the same question using a nested procedure with access to its most local state, I.E. a closure.

Aggregate	Do Not Aggregate

`N = 8`
`solutions = list()`
`search Queen(0, 0):`
`    children:`
`        …`
`    accept:`
`        if len(solution) == N:`
`            append(solutions, solution)`
`            return True`
`        return False`
`    reject:`
`        …`
	`N = 8`
	`search Queen(0, 0):`
	`  children:`
	`    …`
	`  accept:`
	`    return len(solution) == N`
	`  reject:`
	`    …`

Using a closure, how to aggregate solutions and where the data resides is obivous.

There is one, obvious, and non-controversial way to accomplish the task. Anything more would likely be derided during a code review as being over-engineered.

Indeed, the ability to inject state into the behavior with little ceremony was a boon to this very research. Many of the traces and visualizations provided in the *Results* were built simply by capturing a local reference to a file object and constructing a valid *dot* file on the fly from within the search algorithm.

Function-Like Statement Blocks

During the design phase it was an option that our user provided procedures follow an expression-oriented paradigm, similar to that of the Rust programming language, or as seen in Ruby's implicit return.

```
fn main() {
      let x = 5u32;
      let y = {
            let x_squared = x * x;
            let x_cube = x_squared * x;
            // This expression will be assigned to `y`
            x_cube + x_squared + x
      };
}
```

Example Rust code showcasing assignment from a code block. [7]

This would facilitate a more declarative style of procedure. Consider the following examples which show an accept whose body is written as though it were a function and one that is written in an expression-oriented fashion, I.E. it's final expression is returned as the final value for block.

Procedural	Expressional
accept: **return** `len(solution) == N`	**accept:** `len(solution) == N`

Explicit return versus implicit final expression return.

The second, expression-oriented, piece of code feels natural and is easy to pronounce. "This graph search is accepted if the number of accepted nodes is N". The first, function-like, piece of code is not unnatural, per se, but it does beg the question of why we are saying the word return outside of a function body.

It was our preference that these blocks take the expression-oriented route, as we personally found it to be more expressive. However, none of the candidate target languages supported this type of implicit return. Adding this feature to the target language was deemed out scope of the research, both in terms of time and in reach.

As such, the function-like syntax was adopted for the final compiler. In fact, as is touched upon in the *Compiler section* these user provided blocks of code get inserted into a function node of the target abstract syntax tree. Therefor, they look and act like functions because they are, indeed, functions.

Grammar

Finally, we propose the following construct for implementation in an imperative/procedural programming language in Extended Backus-Naur Form.

```
StmtBlock = Inherited from the parent language;
SearchStmt = "search", expression, ":", SearchBlock;
SearchBlock = ChildrenBlock | ChildrenBlock, RejectBlock |
RejectBlock, ChildrenBlock | ChildrenBlock, AcceptBlock | AcceptBlock,
ChildrenBlock | ChildrenBlock, AcceptBlock, RejectBlock |
ChildrenBlock, RejectBlock, AcceptBlock | RejectBlock, ChildrenBlock,
AcceptBlock | RejectBlock, AcceptBlock, ChildrenBlock | AcceptBlock,
RejectBlock, ChildrenBlock | AcceptBlock, ChildrenBlock, Rejectblock;
ChildrenBlock = "children", ":", StmtBlock;
AcceptBlock = "accept", ":", StmtBlock;
RejectBlock = "reject", ":", StmtBlock;
```

EBNF of the search statement.

The following is an example NQueens of 8 solution solved using the above grammar.

```
N = 8
search Queen(0, 0):
    children:
        for row in range(1, N+1):
            yield Queen(node.Column + 1, row)
    reject:
        for queen in solution:
            if (node.row == queen.row or
                   node.column == queen.column or
                   node.row + node.column == queen.row + queen.column
                   node.row - node.column == queen.row - queen.colum):
                return True
            return False
    accept:
        return len(solution) == N
```

Example NQueens solution.

This grammar, and the provided example, will change gradually as we implement it into extant language - technical difficulties and style matching come into play. However, it is representative of our current goal.

Transpiler

Transcription from a source language to another, already well established, target source language is a common practice for prototyping emergent programming languages. The very first Go programs where transpiled into C before being compiled into an executable. [8] The very first C++ compiler, CFront, depended heavily on translations into C. [9] As a proof of concept, we will do likewise for the search construct.

The target output of this transpiler is intended to be a quickly constructed proof-of-concept only. In fact, the implementation of the transpiler has several technical limitations. This includes the inability to rearrange the order of the children, accept, and reject procedures as well as the inability to recursively define a search within a search. These limitations are an artifact of the transpiler implementation and not of the construct itself. In fact, these limitations naturally fade away once properly implemented in a compiler.

The Engine

The following is psuedo code of our target implementation. It will be our goal to transliterate this these semantics into a target programming language.

```
// User definitions //
//////////////////////
let root = FCG
// Our user provided procedures. children is a mandatory
// procedure, and thus can always be assumed to be present.
// If either the reject or accept procedures are left out by
// the user, then these procedures default to returning
// false in all scenarios (I.E. if no subtree is either rejected
// nor accepted, then a full DFS is conducted).
let USER_children function(node, solution) → iterable
let USER_reject function(node, solution) → boolean if provided : false
let USER_accept function(node, solution) → boolean if provided : false
```

```
// Supplemental procedures //
///////////////////////////
// next implements the iterator pattern.
// Returns null on an empty iterator.
let next function(iterable) → node : null
// len implements finding the length of an iterable.
let len function(iterable) → integer
// pop implements a pop from a stack. It is safe
// in that a pop from an empty stack returns null.
let pop function(iterable) → type : null
// push implements the push to a stack.
let push function(iterable, type)

// Engine initialization //
///////////////////////////
let children iterable<node>
let candidate node
let stack list<(node, iterable<node>)>
let solution list<node>

children = USER_children(root, solution)
while true:
  candidate = next(children)
  if candidate == null:
        if len(stack) == 0:
               // Engine termination.
               // No further items in the stack, thus
               // all possible subtrees have been considered.
               break
        // Remove the most recently accepted node
        // from the solution. This will be the parent whose
        // children are currently under consideration.
        pop(solution)
        // Select the last known good sub-tree by popping
        // from the stack.
        root, children = pop(stack)
        continue
  reject = USER_reject(candidate, solution)
  if reject:
        // candidate node rejected out-of-hand. There is no need
        // for data structure updates as rejection is checked
        // before stack or solution manipulation.
        continue
  // If a candidate is not rejected then it is a member
  // of a potential solution.
  push(solution, candidate)
  accept = USER_accept(candidate, solution)
  if accept:
        // If a solution is accepted, then remove it
        // from the solution and continue to search for more solutions.
        pop(solution)
        continue
  // Push the current root and its children to the stack
  // for future consideration.
  push(stack, (root, children))
  // Recurse the structure by making the candidate the
  // new root and receiving its children iterable.
  root = candidate
  children = USER_children(root)
```

Implementation psuedo code of the search statement engine.

Python

Much of our executable code examples written during the discovery phase were in Python. This was due to a combination of a strong support for imperative/procedural programming, first class functions in order to facilitate a higher order function, a lack of type checking which allowed for rapid prototyping, and the general familiarity and comfort that the authors had with the language. As such, it was natural that Python was the first target language of choice.

However, ultimately, Python was not chosen as the final target language. For the sake of thoroughness and understanding, this section will attempt to explain how Python was discovered to be inappropriate for our uses.

Python Closures

In Python closures are *weak*. That is to say, data that is captured by a closure is *read only*.

```
def curryIncrementedAdd(left):
    # Adds left to right, then increments
    # left for the next call to the procedure.
    def closure(right):
       ret = left + right
       left += 1 # illegal write
       return ret
    return closure

add = curryIncrementedAdd(5)
print(add(4))

>> UnboundLocalError: local variable 'left' referenced before
assignment
```

Weak closures in Python disallow for writes to the closed data.

A stated goal of the research was to give the user provided procedures natural, and powerful, scoping rules. Clearly, simply translating these procedures into Python closures will not work. We do, however, have recourse available to us. Python provides the builtin functions, globals and locals, which return a reference to the dictionary used by the Python interpreter to handle global and local scoping. As such, it is possible to pass, as an argument to a subroutine, a reference to one’s local variables.

```
def caller(name):
    callee(locals())

def callee(l):
    print(“callee:“, l["name"])

caller("Guido")

>> callee: Guido
```

Referring to a name local to the parent caller.

However, updates to this dictionary do not take effect in the parent routine.

```
def caller(name):
    callee(locals())
    print(“caller”, name)

def callee(l):
    print("callee:", l["name"])
    l["name"] = "Van Rossum"
    print("callee:", l["name"])

caller("Guido")

>> callee: Guido
>> callee: Van Rossum
>> caller: Guido
```

Writes to the passed namespace do not reflect in the parent caller.

In addition, the names passed to the subroutine are not available in the subroutine as bare names, I.E. they must be referred to via a dictionary lookup.

There is one further option available to us - a technique wherein we use the builtin exec function to execute an arbitrary string as code, which allows us to do consequent updates to the local namespace, thus allowing us to refer to bare names.

```
def caller(name):
    callee(locals())
    print("caller:", name)

def callee(l):
    exec ("") in locals(); locals().update(l)
    print("callee:", name)
```

```
caller("Guido")

>> callee: Guido
>> caller: Guido
```

Referring to a bare name local to the parent caller.

However, name is once again read only in the callee namespace. Additionally, this technique only works with Python 2 - support for it is broken in Python 3.

Already, the complexity and instability of this technique has far exceeded the tolerable threshold for this research. As such, further extrapolation of the idea was abandoned.

Inlining

A second option was to inline the user provided code into the engine. That is, replace every call of a user provided procedure with a block of actual code.

Source	Inlined

`accept:`
`      return len(solution) == N`
	`accept = false`
	`# Enter user land.`
	`accept = len(solution) == N`
	`# Exit user land.`

The accept block transpiled to inline code.

We simply replace return statements with assignments. However, this completely breaks down in the presence of multiple, or early, returns.

Source	Inlined

`accept:`
`      if len(solution) == N:`
`            print(solution)`
`            return True`
`      return False`
	`accept = false`
	`# Enter user land.`
	`  if len(solution) == N:`
	`    print(solution)`
	`    accept = True`
	`  accept = False`
	`# Exit user land.`

Early returns cause incorrect semantics while inlining code.

In the provided case, the accept block will *never* result in a final assignment to True. This is an artifact of the fact that a return is a type of tightly controlled goto statement. In order to achieve its equivalent we require access to the same functionality as a goto.

Source	Inlined
`accept:`	`accept = false`
`      if len(solution) == n:`	`# Enter user land.`
`            print(solution)`	`  if len(solution) == N:`
`            return True`	`    print(solution)`
`      return False`	`    accept = True`
	`    goto END_ACCEPT`
	`  accept == False`
	`  goto END_ACCEPT`
	`# Exit user land.`
	`label END_ACCEPT`

In order to facilitate early returns in inlined code, we must have access to a goto-like construct.

This, however, is not possible in pure Python. It may be possible to modify a given Python interpreter to accommodate this, however the point of a transpiler is to rapidly prototype a proof-of-concept before taking on more complicated work on a compiler or interpreter. As such, such an avenue was deemed inappropriate.

The failures to inject user provided code as either a closure or as an inlined block led us to quickly abandon the use of Python. In its place is a pivot to the Go programming language.

Go

We will attempt to pivot quickly away from Python and instead into the Go programming language.

Go (commonly referred to as Golang in order to prevent confusion with the popular board game as well as to yield more relevant results in search engines) is a statically typed, statically linked, garbage collected, concurrent, imperative/procedural programming languages with support for functional programming as well as a bare-bones OOP.

Go has several characteristics that were very conducive to this research. The stated issues that we had with Python do not apply to Go as Go has both strong closures as well as a, not often encouraged, goto and label system - we could realistically chose to do either-or. It is primarily an imperative/procedural programming language which makes it fit in well with the desires of the research, while simultaneously supporting OOP just enough that a comfortable amount of convenience is provided without being too distracting. It is also an incredibly small language (relative to a language such as C++). In fact, it has such few features that the addition of a search statement yields fairly predictable interactions with the language's extant features.

Finally, the Go compiler is written in Go and the authors are intimately familiar with programming in Go. Meanwhile, the authors are not particularly experienced in working with very large C projects, such as the CPython interpreter. This familiarity made the future task of compiler modification a much more tractable problem within the time constraints of the research.

A Brief Introduction

At the time of writing, Go is a new enough programming language that we believe that it is worthwhile to very briefly explain at least several key concepts and terminology that will be required to continue on.

1. Functions are first class.
2. Lambdas are assignable to variables.
3. A slice is a built in, resizable, linear collection.
4. A goroutine is a lightweight thread managed by the Go runtime.
5. A chan is a reader/writer queue used for communication and, optionally, synchronization between goroutines.
6. The := operator performs simultaneous implicit declaration and initialization, such that lhs := rhs is the statement that, “lhs is declared to be a variable whose type and initial value

is the result of the evaluation of rhs”. The statements x := 5 and var x int = 5 are semantically equivalent.

7. The built in function make allocates and initializes the three primitive data structures in Go - the slice, the map, and the chan.
    a. make([]int, 0, 5) // a slice of ints with a length 0 and an optional capacity 5
    b. make(map[int]string, 5) // a map of ints to strings with an optional capacity 5
    c. make(chan int, 0) // an unbuffered channel of ints
8. It is idiomatic for the verbosity of a variable name to be a function of its importance and scope. Consequently very local, ephemeral, variables tend to have single character names. The authors acknowledge that this may not be to everyone’s taste and will attempt to keep this style in check.
9. The defer keyword ensures that the deferred function call is always executed upon the exit of the current function. E.G.
    a. func talk() {
    b.   defer fmt.Println(“Second”)
    c.   fmt.Println(“First”)
    d. }
10. The <- operator puts into and receives from a channel. The position of the operator is a mnemonic for how it is behaving.
    a. channel <- value // put into channel
    b. value := <-channel // receive from channel

We expect that these clarifications will be enough such that the C-like nature of Go will allow for the reader to fill in any further gaps.

Structure, Block Delimiters, and Aesthetics

With the switch to Go, the aesthetics of the search construct changes slightly. Go is a C-like language with the following rules.

1. Statement blocks are delimited with { and }.
2. The only exception to 1 are the case and default statements within a switch, which are delimited with a : and closed either by another case, default, or the } which closes the switch itself.
3. Statements are terminated by a ;. However, in any case where a statement may legally end and a newline is encountered, the compiler inserts an implicit semicolon.
4. Indentation level is not significant.

Using these rules, we will model our search statement off of Go's switch statement. The idiomatic style in Go for a switch is to have each individual case at the same indentation level as the switch itself, and as such we will do the same for search.

```
search FCG {
children:
        …
accept:
        …
reject:
        …
}
```

The search statement grammar implemented in Go.

Addressing the Type System

Python's dynamic typing allowed for rapid prototyping as no type system had to be considered. In Python, the solution list was comprised of objects whose type is that returned by the children procedure - whatever that may be. However, we must now work with Go's type system, such that the following hold true.

1. The given target type is called T.
2. The FCG is of type T.
3. The solution data structure is a slice of type T.

4. The children procedure accepts a a type T and a slice of type T and returns a channel of type T.
5. The reject procedure accepts a type T and a slice of type T and returns a boolean.
6. The accept procedure accepts a type T and a slice of type T and returns a boolean.

As of Go 1.10 the language has no support for generic/templated programming. The runtime has facilities for reflection, however in Go types are not values. That is to say, something like the following is not possible.

```
// Runtime information about a type is available,
// however this cannot be used to, say, declare
// or initialize new variables and structures of that type.
t := reflect.TypeOf(FCG)
solution := make([]t, 0) // Will not compile
children := func(node t, solution []t) bool {} // Will not compile
```

In Go, types are not values, disallowing the use of reflection to simulate generics.

Since a transpiler is merely a text processor, no type information can be accurately inferred. This forces us to *ask* the user explicitly what type they are referring to. For this, we have chosen to include the name of the user defined type as the second field in the search signature.

```
search FCG; Type {
children:
      …
accept:
      …
reject:
      …
}
```

The explicit name of the target type is to be placed in the search statement signature.

The second field in the signature, the target type, can then be interpolated into the target source text where necessary.

This is unsatisfactory, and technically unnecessary in the core context of the research, but necessary for the moment. We will attempt to remove this technical blemish when working in the compiler, as type information should be then available. However, for now, it must stay.

Extraction and Text Processing

Language parsing requires, of course, a parser. However, to write a parser for a transpiler is to write a smaller compiler whose output is piped into a larger compiler. While this is fine, and indeed desirable for longer term solutions, the goal of this stage of the research is to rapidly produce a proof that core engines works and that, at least in general, the desired syntax is workable.

When one thinks of rapidly producing text processors a natural first instinct is, "can I solve this using regular expressions". Surprisingly, in this case, yes although it will take a single pass of pre-processing to do so. Let us consider the skeleton of the search statement.

```
…
search FCG; Type {
children:
      …
accept:
      …
reject:
      …
}
…
```

A statement list,, followed by a search statement, followed by a statement list.

The above, of course, is not a regular language. Immediately apparent is the archetypal balanced-parenthesis problem. However, the balanced-parenthesis problem is really our only barrier-to-entry to solving this problem using a regular expression. That is, the signature is a regular language (search followed by any text, followed by a semicolon, followed by any text, followed by a left curly brace) as are the individual procedure blocks (the given keyword, followed by a colon, followed by any text in a non-greedy fashion). The only trouble is how to tell if a given right curly brace is the terminal brace for the search statement. Therefore, we must solve the balanced-parenthesis problem first, then supply the output to a regular expression engine. We will accomplish this by isolating the target block of source code in the following fashion.

1. Consume consume characters until a search signature is found.
2. Begin counting curly braces.
3. While curly braces are unbalanced, consume characters and place them into an intermediate buffer.

Once this first pass over the text completes, we will have the following in an intermediate buffer.

```
search FCG; Type {
children:
      …
accept:
      …
reject:
      …
}EOF
```

After a single pass of the search statement, it becomes a search statement followed by an end-of-file.

That is, by balancing the curly braces once we no longer need to look for a balanced closing curly brace for the search statement - we need only look for }EOF.

We can now construct a regular expression that can capture the appropriate groups (the FCG, the node type, and the procedure blocks) from which we can simply perform a string interpolation to produce a block of pure Go source code.

Merely for the sake of curiosity (and perhaps a fascination of the abomination), the following is in fact the regular expression that captures our construct. Its syntax targets the Go standard library `regexp` package.

```
(?sU)\s*search\s+from\s+(?P<FCG>.*)\s+;(?P<UTYPE>.*)\s*{\s*children\s*
:\s*(?P<CHILDREN_BODY>(?:.*)*)\s*(?:accept\s*:\s*\s*(?P<ACCEPT_BODY>(?
:.*)*))?\s*(?:reject\s*:\s*(?P<REJECT_BODY>(?:.*)*))?\s*}$
```
Regular expression of the search statement.

Scoping

Our transpiled engine does contain names used by the engine itself that are not intended for consumption by the user. We shall use Go's scoping rules in order to prevent the user from accidentally, or intentionally, clobbering or violating names not meant for them.

In Go, local scopes have read/write access to all names declared within scopes higher than itself, up to the maximum of the package level scope.

```
number := 0
if true {
      number = 1 // assignment with no declaration
}
log.Println(number) // 1
```
A lower scope writing to a name declared in a higher scope.

Names, however, can be shadowed in local scopes. That is, the same name can be redeclared in a scope without affecting the more global name.

```
number := 0
if true {
      number := 1 // new declaration and assignment via := operator
      log.Println(number) // 1
}
log.Println(number) // 0
```
A new declaration shadowing the declaration of the same name in a higher scope.

Additionally, declarations in lower scopes do not persist in consequent higher scopes.

```
if true {
      bob := "bob"
      log.Println(bob) // bob
}
log.Println(bob) // Compiler error, name not found
```

Names are limited to their most local scope.

We will leverage this by first placing all user defined procedures at the very top of the `search` engine so that they do not have access to names declared below them. Second, we will place the entire engine in an unconditionally true block.

```
if true {
      children := func() {...}
      accept := func() {...}
      reject := func() {...}
      root := FCG
      ...
}
```

Scoping the entire search statement engine by enclosing it an unconditionally true block.

This prevents any consequent user code from accessing internal names. Additionally, it quickly nominates all names defined with the scope for garbage collection immediately after the search terminates.

Results

The transpiler successfully translates, writes to disk, and submits to the Go compiler a valid Go program. Examples of such programs will be provided in the *Idioms* as we discuss working programs and emergent idioms.

This transpiler, however, is not a satisfactory end to the implementation of the `search` statement. Downfalls include:

1. A lack of the type inferencing required to discover the type of the FCG.
2. The children, accept, and reject blocks *cannot* be rearranged in any order that the user likes.
3. A transpiler precludes any research that can be done on compiler code optimization.

4. User code is difficult to debug due to differing line numbers in the source code and the transpiled code.
5. It is significantly slower and more cumbersome than the Go compiler itself.

Compiler

The Go compiler shall be our target workspace. We will be working off of the Go 1.10 release.

Abstract Syntax Tree Splicing

Via our transpiler we have shown that the search statement semantics can be represented at the highest order of abstraction - pure source code. That is, no handcrafted assembly, nor special linking, nor any other tactic that is so close to the CPU is required. As such, we would prefer to work in the highest abstraction that we can while still providing native compiler support. Therefor, our target change to the Go compiler will lie in its abstract syntax tree representation. In particular, we will take advantage of the fact that the search statement engine is static code which merely requires that the user provide several facts and procedures to fill in the necessary gaps. We shall accomplish this by copying a static AST that is representative of the engine, swapping in nodes provided by the user, and splicing the resulting subtree into the target AST.

First, we begin with an unmodified AST which is comprised of some code, followed by a search node, followed by some code. The search node is comprised of an FCG node, a children node, a reject node which may be nil, and an accept node which may be nil.

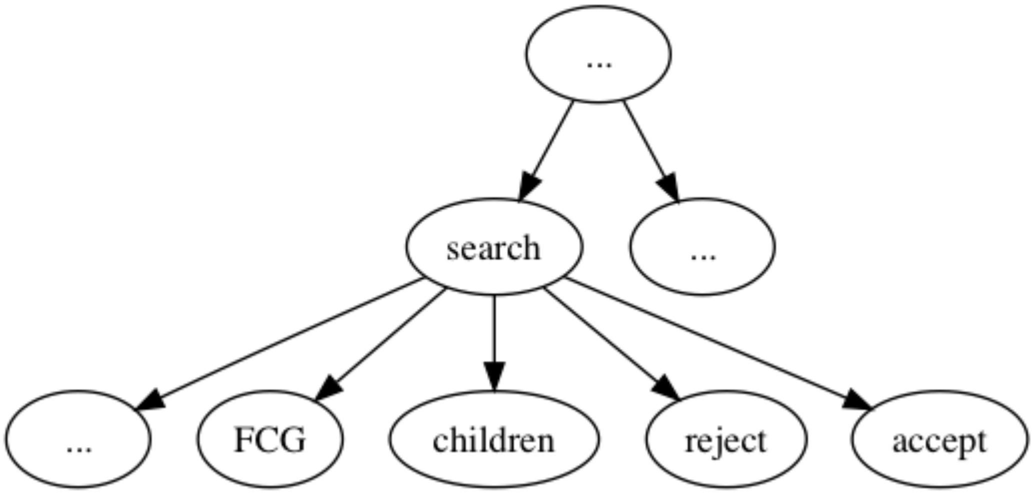

Figure 12: An AST with a search statement somewhere within the tree.

The FCG node is some expression, and each of the user provided procedures are to be treated as function bodies. In source code, we can represent the preamble to the search engine as follows.

```
root := FCG
// User CHILDREN declaration.
USER_children := func(node TYPE, solution []TYPE) <-chan TYPE {
    BODY
}
// User ACCEPT declaration.
USER_accept := func(node TYPE, solution []TYPE) bool {
    BODY
}
// User REJECT declaration.
USER_reject := func(node TYPE, solution []TYPE) bool {
    BODY
}
```

The target code where user provided code is to be injected.

That is, we must fill in the appropriate nodes to complete the AST representation of the above code.

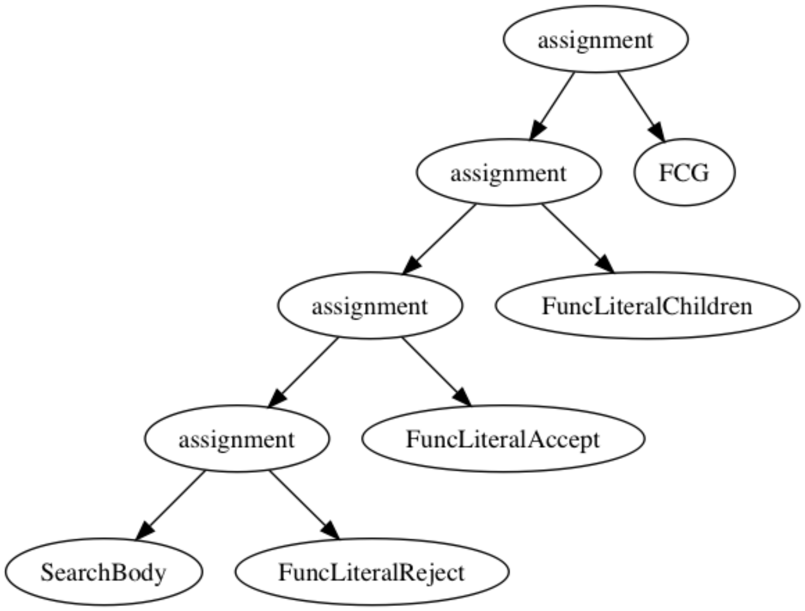

Figure 13: The constructed AST of the search statement engine.

Once we have constructed this isolated abstract syntax tree we can splice it back into the original AST where the search node use to reside.

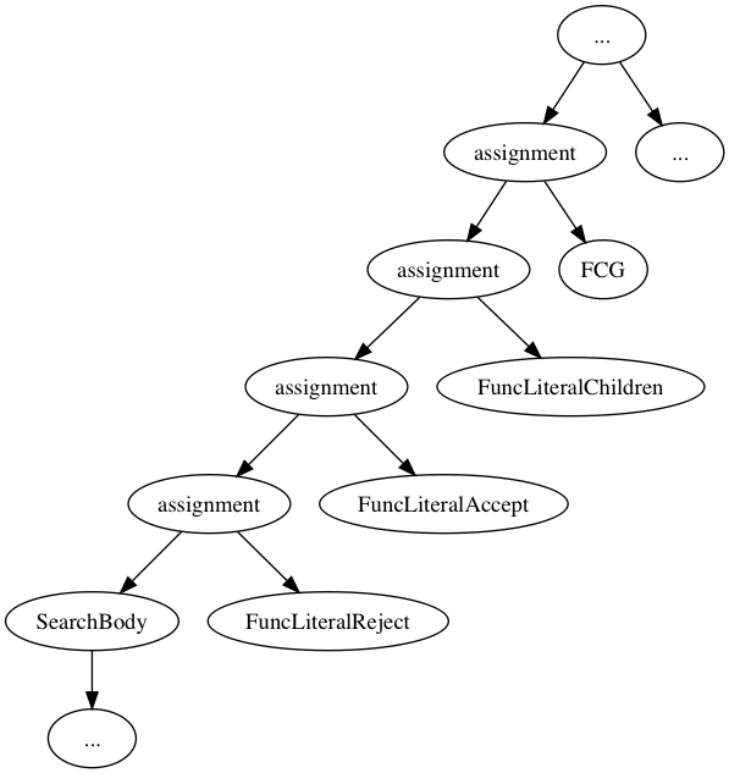

Figure 14: Splicing the completed search statement engine back into the target AST.

From hereon we must only concern ourselves with the normal actions of linking the first statement of the search block to the statement above it, and linking the final statement of the search block with the statement below it. Any other work is unnecessary. The user provided nodes are stemmed from pure Go as is the preconstructed, spliced, AST. This means that any guarantees already provided by Go are provided by our implementation as well.

Scoping

In the transpiler we forced to use our knowledge of Go's scoping rules in order to protect engine internal names from malicious, or unlucky, users. We accomplished this by enclosing the entire engine in an unconditionally true block and placing all user provided definitions at the very top of that block.

```
if true {
	children := func() {...}
	accept := func() {...}
	reject := func() {...}
	root := FCG
	...
}
```

The search statement engine scoped by an unconditionally true block.

This is unnecessary now that we are working in the compiler. The Go compiler code has facilities for opening and closing scopes wherever one pleases. Therefor we can simply open a new scope for the engine and forego our hack in the transpiler.

The Type System

Our transpiler was merely a text processor. As such it could not have the deeper understanding of the type system that is required to infer the type of the FCG. This left us working with the following signature wherein we explicitly ask the user to provide us with the type information.

```
search FCG; Type {
children:
	…
accept:
	…
reject:
	…
}
```

The search statement with a required named type in the signature.

We will now attempt to remove this blemish.

The construction of the Go AST is a two pass algorithm. We will refer to these passes as the first past AST and the second pass AST.

The first pass AST is constructed during the consumption of the source programs tokens. This stage in the compiler tracks only the most basic information such as the original line of source code as well as the original text. Errors in this stage are very forgiving - only basic syntactical errors are detected and emitted. As tokens are consumed, their structure is checked against the languages syntactical rules and added as nodes to the target AST. Once all tokens are consumed, this first pass AST is complete and given as input to the second pass AST.

The second pass AST is responsible for construction of the intermediate language emitted to the assembler as well as enforcing the semantic rules of the language. Dependency resolution (packages and names), import and exportation rules, the type system, and so on. For example, if a name is detected to have been declared, but never used, this is the stage in the compiler which emits the appropriate error. Such errors are detected, and emitted, immediately as they are discovered as the second pass AST consumes the first pass AST node-by-node. As such, any modifications we wish to make to the intermediate representation which corrects errors in the type system must occur *before* the second pass AST. I.E. we must work in the first pass AST to accomplish our goal.

A valid FCG can take on many forms. It can be a reference, a pointer, a struct literal, a slice/array literal, a map literal, a channel, or even a function or lambda. This list is not comprehensive. The FCG can be anything in Go which is a valid expression. The extraction of this particular node from the AST is not difficult, relative to the search keyword it is deterministically in the same location each time. However, at this moment in the compiler there is no deeper understanding of the node’s type other than textual information - this deeper understanding will only come later during the second pass AST, at which point it is too late. The textual information *does* contain what appears to be necessary to infer what type the expression will evaluate to. However, given all that an expression can be, we could not identify a simple way to accurately predict the type without actually creating a quarantined version of the type checker itself.

Without access to help from someone with working knowledge of the Go compiler we had to abandon our attempts to infer the type of the FCG due to time constraints. As such, it is an unfortunate failure of engineering that users must still provide the compiler with what type the FCG evaluates to.

Perfect Hashing

In order to facilitate the rapid lookup of reserved keywords with a provable access time, the Go compiler constructs a custom hash table of all of the language reserved keywords using a perfect hash for the given set. The assumptions made by the Go team's perfect hash were that A). every keyword has at least two characters and that B). no two keywords in the language share a common first two characters.

```
func hash(s []byte) uint {
    return (uint(s[0])<<4 ^ uint(s[1]) + uint(len(s))) & uint(len(keywordMap)-1)
}
```

The perfect hashing function for reserved keywords in the Go compiler.

This, however, is no longer true. Trivially we can see that this does not hold for search and select, children and chan, and reject and return. Our immediate solution to this problem was to keep the core hashing algorithm, but to select different seconds characters for our own keywords.

```
func hash(s []byte) uint {
    var second uint
    switch string(s) {
    case "search":
         second = uint(s[3])
    case "children":
         second = uint(s[6])
    case "reject":
         second = uint(s[2])
    default:
         second = uint(s[1])
    }
    return (uint(s[0])<<4 ^ second + uint(len(s))) & uint(len(keywordMap)-1)
}
```

The perfect hashing function for reserved keywords in the Go compiler after adding the search statement.

Keyword Reservation and Toolchain Bootstrapping

It was taken for granted that in our new compiler the words search, children, accept, and reject are now language reserved keywords. That is, they are no longer valid variable, type, nor function names. Given the fact the Go compiler is written in Go, and that the Go compiler utilizes the Go standard library, we had to contend with the fact that immediately upon reserving a word in the language that the language itself would fail to compile as many tens-of-thousands of extant source code files used those names freely.

search was found frequently in the source code of the implementation of Go’s map type, which is a type of associative array implemented via a hash table and thus used the word search in its probing algorithm for handling hash collisions. children was used rarely in the toolchain, although there are several custom implementation of various trees, including a red-black tree, which used the word. accept is incredibly common in Go’s networking stack. Finally, the most fascinating word was reject which is said not even a single time in the entire Go toolchain. accept and reject are of course antonyms. reject can just as easily be said as !accept and accept can just as easily be written as !reject - it is merely a matter of preference and outlook. The Go team appears to prefer the former.

The solution to this was the simple pre-pending of every offending word with an underscore.

Concurrency

With the move to Go as the target programming language we suddenly had available to us very strong concurrency primitives as well as a strong incentive - any search down a subgraph is irrespective of the search down any other subgraph. Upon meditation, the user interface for conducting a concurrent graph search can be very simple - simply tell the engine how many concurrent workers you would like to have at any given time. As such, it is fitting for a third, optional, value in the search signature.

```
search FCG; Type; ConcurrencyLevel {
    ...
}
```

The concurrency level, an integer, as a part of the search statement signature.

The number of workers provided to the search signature must be a positive integer greater than 0. A value of 0 will successfully compile, however semantically it is stating that, “I would like to conduct this search using exactly no workers", which will immediately result in a deadlock at runtime. If a negative number is provided statically (I.E. a constant or literal) then the program will not compile. If a negative number is provided dynamically (I.E. a call to a function that returns an integer) then the program will crash during runtime.

Goroutine A searches a graph starting at the given FCG and an empty solution. Upon discovery of a node that is *not* rejected and that is *not* accepted (I.E. a candidate member of a valid solution, but not the terminal member of a solution) the engine checks for the availability of any worker routines. If no workers are available, then goroutine A continues on normally. If a worker routine is available, then the solution thus far is copied and a new goroutine, B, is created. Goroutine B is initialized with the copy of A's solution structure as well as the newly discovered node acting as the FCG. Goroutine A then continues on as though the newly discovered child had been accepted/rejected (I.E. it continues on with the next available child).

```
if !reject(node, solution) && !accept(node, solution) && workerAvailable {
    newFCG := node
    subsolution := make([]Type, len(solution)
    copy(subsolution, solution)
    subsolution = append(subsolution, node)
    go engine(subsolution, newFCG)
}
```

Control flow of the execution of a new worker.

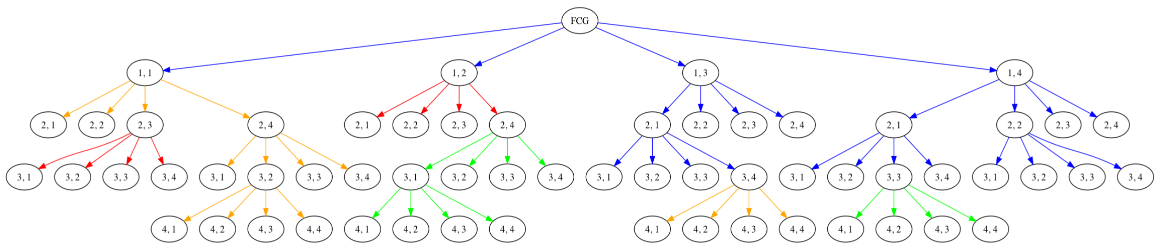

Figure 15: An actual trace of the NQueens of 4 problem. Each color represents a different worker solving a subgraph. Notice how a given worker (orange, red, and green in this case) can jump around to solve different parts of the graph as they become available.

RESULTS

Idioms

Children Generator Goroutines

The interface of the children block simply states that an *iterable* of a some kind must be returned for consideration in the engine. What this iterable exactly *is* was left to the discretion of the user when writing in Python. However, now that we are working in a static type system, we must statically consider what this type is.

It may be possible to allow the user to return one of any number of different types which can be used iteratively. Target, builtin, types are the slice, the map, and the chan. Any of these types can be detected in the compiler and accommodated for in the engine. However, such a tactic faces the exact same engineering troubles that were faced when we were attempting to work with the type of the FCG - modification of placeholder type information can only be done after type checking has occurred at which point it is too late. As such, we decided to choose to allow for only type of iterable - the chan.

The chan was ultimately selected out of concerns for efficiency. Given an unbuffered channel of children, the space complexity of the return of the children block is O(1). That is, at any given time there is one child that is being considered as a subsolution, and one child that is waiting to be put into in the channel.

```
children:
    c := make(chan int, 0)
    go func() {
       defer close(c)
       for child := node + 1; child < MAX; child++ {
            // Waits indefinitely until a receiver asks for a value.
             c <- child
       }
    }()
    return c
```

The generator pattern as implemented in Go used to generate children.

If we were to select a slice or a map instead then the children block would be required to precompute all children. For the children block this leads to a size complexity of O(N) where N is the number of children a given node has.

```
children:
    c := make(chan int, 0)
    for child := node + 1; child < MAX; child++ {
        c = append(c, child)
   }
   return c
```

The children block returning a slice of children rather than a channel.

However, as we traverse the graph children iterables are pushed to the stack alongside with their parent nodes. Therefor, if all children for a given node are precomputed then the space complexity for keeping track of all precomputed children is O(H * N) where H is the maximum height of the graph and N is the number of children per node. If instead all children are generated as needed, then the space complexity is O(H).

The generator pattern is, however, more complex in code than simply returning a slice. A goroutine *must* be used to asynchronously feed children into the channel. Any attempt to place a value into the channel from within the parent goroutine will result in a deadlock.

```
children:
    c := make(chan int, 0)
    defer close(c)
    for child := node + 1; child < MAX; child++ {
      // Deadlock! This code will never proceed as it is waiting
      // on the engine to receive from the channel that has
      // not been returned yet.
      c <- child
    }
    return c
```

A deadlock caused by the improper handling of an unbuffered channel.

This may seem perilous at first, however this style of generator is incredibly common and, in fact, idiomatic in the Go programming language. Go programmers are extremely comfortable with the presented pattern, as well as working with goroutines and channels, both buffered and unbuffered. As such, we felt comfortable presenting this pattern as *the* way to handle iteration over children.

An Empty Graph

An empty tree is easy to represent. It is an FCG followed by exactly no nodes.

```
search FCG; Type {
children:
      c := make(chan Type, 0)
      close(c)
      return c
}
```

A search that is always empty.

This is, in effect, a very expensive no-op (the compiler makes no attempt to detect that this graph search will produce no nodes, and as such performs all necessary setup for a task that will in fact lead nowhere).

It is useful, however, when writing a procedure that can accept any FCG to a given graph type and solve that graph. For example, let us consider a simple walk through a given person's social media account, looking for all connections to that person. The root person's account is the FCG to the graph search.

```
func find(FCG Account) {
    search FCG; Account {
    children:
        c := make(chan Account, 0)
        if isolated(node) {
              close(c)
              return c
        }
        go func() {
              defer close(c)
              for connection := range connections(node) {
                    c <- connection
              }
        }()
        return c
    }
}

find(celebrity) // Many millions of connections
find(politician) // Likely fewer than the celebrity, but still many
find(toddler) // An infant will (should) have no connections on social media
```

A search that is conditionally empty.

The toddler has no connections on social media, and thus is marked as being isolated, and thus is the FCG to an empty graph. However, there is an even more natural way to represent this.

```
func find(FCG Node) {
    search FCG; Node {
    children:
        c := make(chan Node, 0)
        go func() {
            defer close(c)
            for connection := range connections(node) {
                c <- connection
            }
        }()
        return c
    }
}
```

A search that is conditionally empty by virtue of having no children.

That is, a node that yields an empty collection of connections is implicitly isolated, and thus is an FCG that yields an empty graph.

Cycle Detection

A question that arose during the research was whether or not it was possible to detect, and prevent cycles on behalf of the user. In terms of implementation it is possible, if inefficient. If the user provided nodes are of a pointer type then it is trivial to detect cycles - simply place known pointers into a hash set. However, if the user provided nodes are of a reference type then deep equality must be performed in order to discover cycles. Given the typical size of graph problems, this seemed rather egregious to chose to do implicitly.

However, efficiency and logistics aside, we decided against automatic cycle detection for the sole purpose that it would be incredibly *surprising* to many developers who did not intend for such a thing to happen. That is, it is not a provable *fact* that just because a cycle has been detected that it must not be followed.

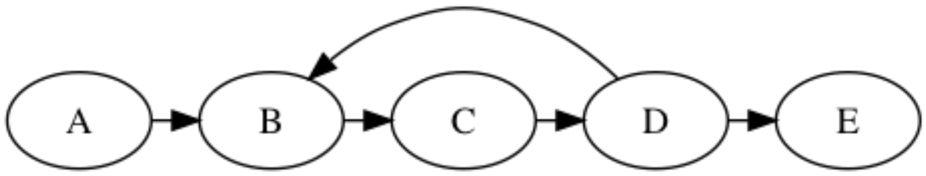

Figure 16: Although it is often avoided, following the cycle from D back to B is not an inherently illegal maneuver.

A toy example is that of an offroad race track. Let us say that the sponsors of an upcoming race would like to use our search statement in order to design a race track which maximizes difficulty as well visibility to live bystanders and camera crews. In order to accomplish this, they would like certain loops within the race track to be ran by the participants at least several times in a row before moving on with the rest of race. If the search engine code automatically, and implicitly, kills computations of these cycles then the developers will be left confused and frustrated as to why none of the desired cycles appear. That is, until they discover that they are at the mercy of the engine, at which point they will most likely abandon the engine itself rather than attempt to fork and modify it.

However, cycle detection is more common than not. Fortunately it is rather easy to construct cycle detection in userland code. Let us consider cycle detection in a webcrawler. We have two choices, either we can detect the cycle as we are generating children, and thus decide to not generate a particular child if a cycle is detected, or we can elect to inject our code into the reject block which will return true if the given URL has been visited before. The accept block is too late as at that point the node has been already been considered valid and appended to the solution.

Children	Reject
`visited := make(map[string]bool, 0)`	`visited := make(map[string]bool, 0)`

```
search seed; string {
children:
    c := make(chan string, 0)
    hrefs := ExtractAllHrefs(node)
    visited[node] = true
    go func() {
        defer close(c)
        for _, href := range hrefs {
            if visited[href] {
                continue
            }
            c <- href
        }
    }()
    return c
}
```

```
search seed; string {
children:
    c := make(chan string, 0)
    hrefs := ExtractAllHrefs(node)
    visited[node] = true
    go func() {
        defer close(c)
        for _, href := range hrefs {
            c <- href
        }
    }()
    return c
reject:
    return visited[node]
}
```

Cycle detecting implemented in either the children block or the reject block.

Either is semantically valid in all cases. During the research it was the preference of the authors to perform cycle detection while generating children, simply for the sake of failing candidates as fast as possible. It can be argued, however, that this notion is a slippery slope - if one's goal is to eliminate choices early then seemingly in all cases the user can forgo the use of the reject block and place the same logic into the children block, thus obviating the reject block altogether. From this perspective, the reject block is the one true location for cycle detection since being a cycle is a type of rejection itself.

Manual Solution Handling

The search engine maintains a single data structure meant for the consumption of the user - the solution slice. However, it may be beneficial for a given user to maintain their own solution structure and forego the use of the engine provided structure. Specific examples of why this would be desirable are covered in the *Scalability* section, but for now let us observe *how* it would be accomplished.

It is at this point that some specific knowledge about the underlying algorithm may be helpful in expediting one's understanding. However, we will attempt to avoid addressing the underlying algorithm directly, instead favoring our desired approach - thinking about the behavior of graphs.

First we will consider when to add a node to a candidate solution. Intuitively, a node may be considered a member of a solution if that node has *not been rejected*.

```
reject:
    if valid(node) {
        solution = append(solution, node)
        return false
    }
    return true
```

Manually constructing a solution upon non-rejection of a node.

Alternatively, one may consider a node a member of a candidate solution if that node has been asked for its children (that is, it has been visited).

```
children:
    solution = append(solution, node)
    c := make(chan Type, 0)
    go func() {
        ...
    }
    return c
```

Manually constructing a solution upon visiting a node.

It is technically possible to append a node to the solution in the accept block, as a call to the procedure is immediately preceded by a return of false by the reject block. However, we will ignore this as it is semantically strange and it leverages knowledge about the underlying algorithm.

Second we will consider when a node should be removed from a solution. The remove operation in this case must be a *pop* type operation. In the case of removal, rather than having two different *options*, we have two different *requirements*. That is, there are two cases where a node *must* be removed - after a solution has been accepted and after a node has completely exhausted all possible children. Removal during rejection does not make sense since at that point the candidate is not even a member of the solution.

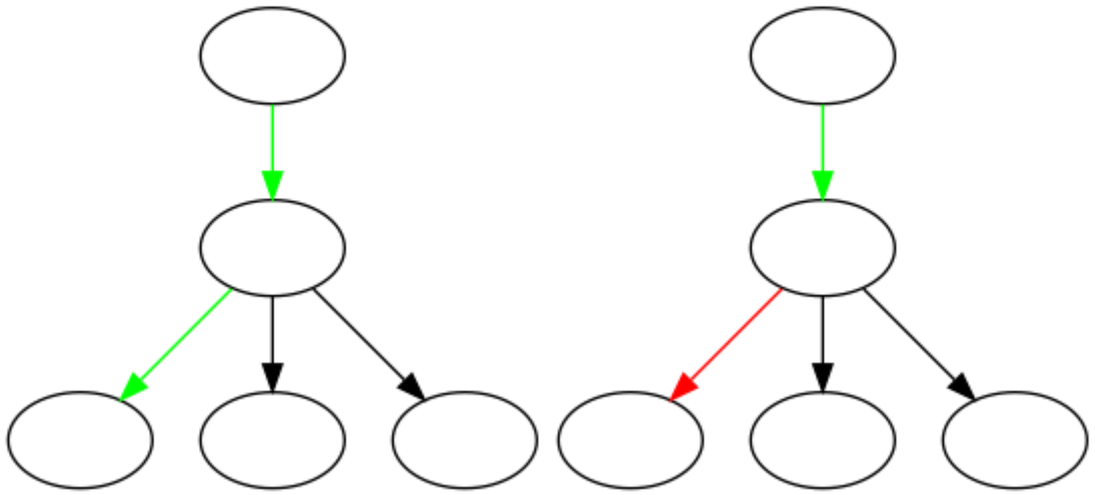

Figure 17: A solution has been accepted. The latest node in the solution must now be removed and the rest of tree considered.

Removal must occur in the event of the acceptance of a solution. Failure to do otherwise will result in a solution structure which aggregates all previous accepted solutions in addition to the currently considered solution.

```
accept:
    if valid(solution) {
        …
        pop(solution)
        …
        return true
    }
    return false
```

Removing a node from the solution if it is the final node of an accepted solution.

Removal must also occur in the event that a parent node has exhausted all possible children. That is, having no children left to consider means that the parent should no longer be considered.

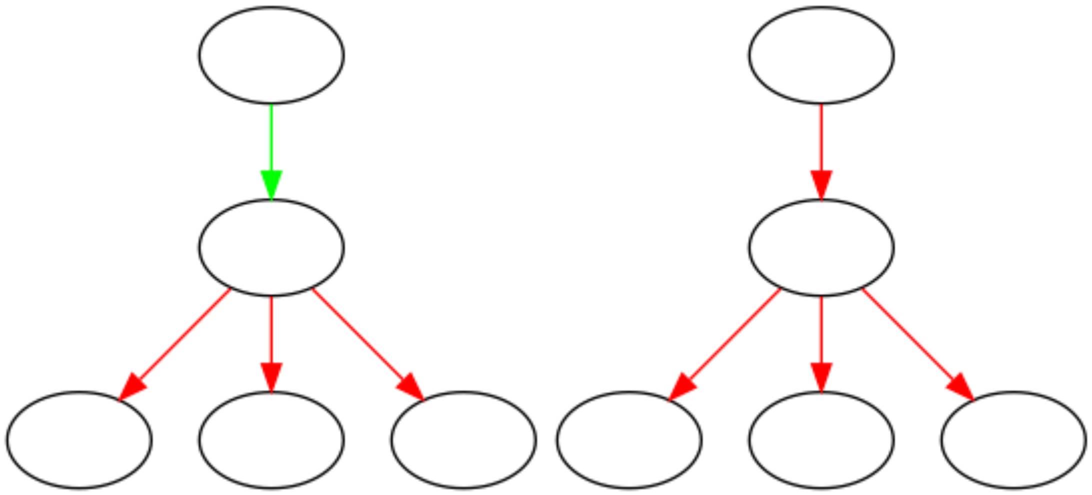

Figure 18: The search of all children have been exhausted. The parent must now also be removed from the solution.

From within the code, removal of the parent has the greatest pitfall that we have encountered in the user code. The following is intuitive, but incorrect.

```
children:
    c := make(chan Type, 0)
    go func() {
        defer close(c)
        for child := range connected(node) {
            c <- child
        }
        pop(solution) // Incorrect early removal
    }
    return c
```

Early removal of a node from a solution. This will remove a parent from a solution before the final child has been fully explored.

The above removes the parent from the solution *immediately* upon submitting a child for consideration. That is, the code is stating that, “the parent is to be immediately removed after all children have been submitted for consideration”. Rather, the code should say, “the parent is to be removed immediately after all children have been considered”. A parent is signaled that a given child has been completely considered once the parent is asked for the next of its children. As such, correct code waits on the ability to send on the channel.

```
children:
    c := make(chan Type, 0)
    go func() {
        defer close(c)
        for child := range connected(node) {
            c <- child
        }
        c <- Type{} // Wait and send some invalid zero value
        pop(solution) // All children have now been considered.
    }
    return c
```

The engine asking for another child is a signal that the previous child has been completely explored.

Using these techniques, it is possible for userland code to manage their own solution data structure.

Breaking Out

It may be desirable for the user to terminate a search early. Most commonly, this would likely be when the user wishes to find only *some* answer rather than *all* answers.

```
accept:
    if valid(solution) {
       break
    }
    …
```

The early break of a search.

However, in our implementation, a break would merely force a return of the block, as it is implemented as a function in the AST of the compiler. Additionally, goto may not refer to a label defined outside of a function. Therefore, as all user defined behavior is implemented as a function, the user has no succinct way to terminate the engine early. Instead, they must manually shutdown the search. A good way to accomplish this is to signal to the children block to stop generating children.

```
var done bool
search FCG; int {
children:
    c := make(chan int, 0)
    go func() {
        defer close(c)
        for child := node + 1; child < MAX; child++ {
             if done {
                 return
             }
              c <- child
        }
    }()
    return c
accept:
    if valid(solution) {
        done = true
        return true
    }
    return false
}
```

Manually handling the early termination of a search by setting a “done” signal.

Concurrency

Concurrent graphs searches do not require any special attention by the user that is unexpected. However, it is worth noting the obvious that a synchronization strategy *must* be used for any data for which the userland code executes a write.

```
var done bool
var lock sync.Mutex
search FCG; int; 8 {
children:
    c := make(chan int, 0)
    go func() {
       defer close(c)
       for child := node + 1; child < MAX; child++ {
            lock.Lock()
            if done {
                lock.Unlock()
                return
            }
            lock.Unlock()
             c <- child
       }
    }()
    return c
accept:
    lock.Lock()
    defer lock.Unlock()
    if valid(solution) {
        done = true
        return true
    }
    return false
}
```
A search conducted using eight worker routines.

The method of synchronization that the user selects is at their discretion.

Scalability

Efficiency

The time complexity of the engine itself, in a single worker context, is a amortized O(1). That is, the engine's computational performance is entirely dependent upon the time complexity of the userland code. It is an amortized O(1) due to the use of Go's builtin slice type for stack and solution management, which is a dynamic array. However, the execution of the engine in a multi-worker context opens the possibility for a higher time complexity. As workers become available the solution to their parent node is copied for consideration in a subgraph. The size of this solution has a maximum size of $H - 1$, where H is the maximum height possible within the search. If the number of workers available to the computation is equivalent to the number of nodes in the graph, then it is possible to incur this copy at every possible point in the graph. This gives a time complexity of $O(H^2)$. The average case, however, is much more difficult to calculate as it is dependent upon three variables - the number of workers available, the exact nature of the graph, and the semi-random behavior of the Go runtime scheduler.

The space complexity of the engine is $2HW$, or $O(HW)$, where H is the maximum height of the graph and W is the number of workers active in the computation. Much like the time complexity, as the number of workers saturate the graph in question, the space complexity trends towards $O(H^2)$.

We provide the following benchmark comparisons. We will solve the NQueens problem from sizes 1 to 15, inclusive. For each size of the NQueens problem we will execute it on 1 to 8 CPU cores, inclusive. Each size/core combination will be executed 40 times. The average execution time and number of memory allocations will be calculated for each size/core combination.

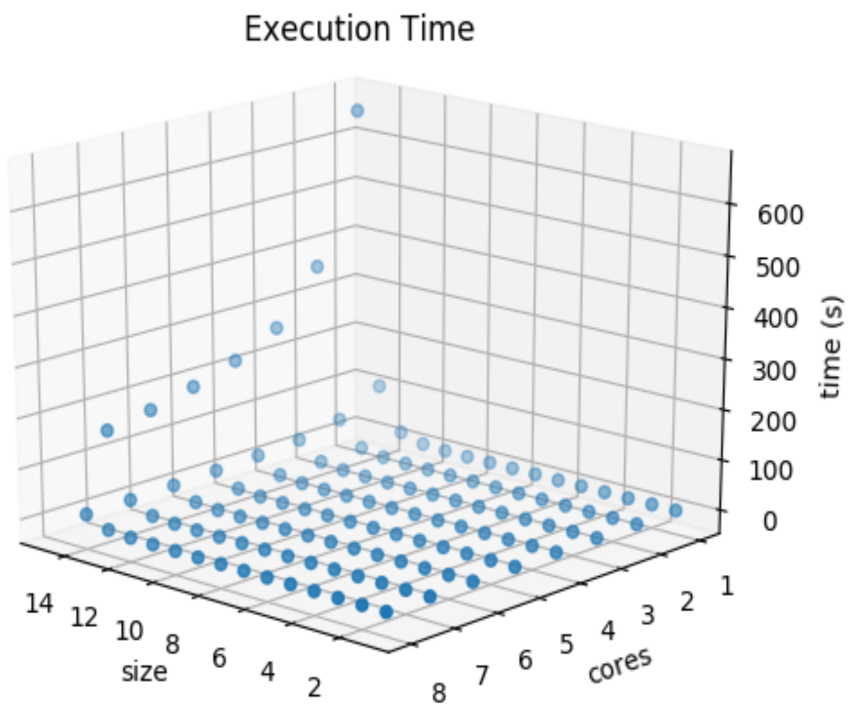

Figure 19: The NQueens average execution time.

Performance gains due to concurrent programming are logarithmic for the NQueens problem. As more cores/workers are added, the more overhead of copying data structures is incurred. Additionally, graphs have an inherent *saturation* level that is dependent upon the number of failure and acceptance nodes within the graph. The higher the number of terminal node, the more likely it is for a worker to be spawned only to be immediately killed and restarted. This saturation level is much less likely to be achieved on something like a large social network where terminal node are rare, however in the NQueens problem terminal nodes with the graph are incredibly common.

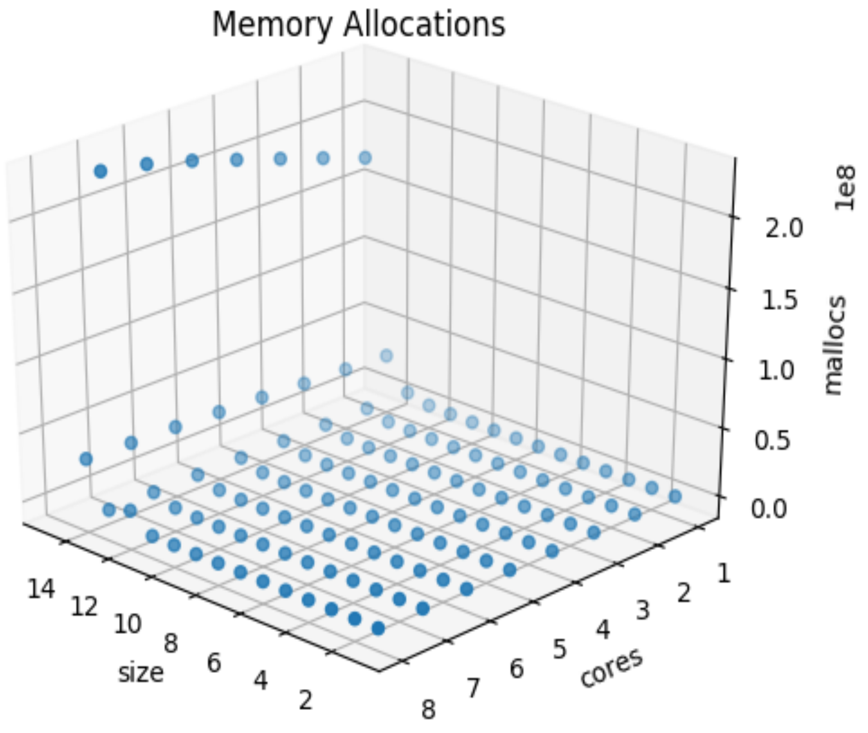

Figure 20: The NQueens average memory allocations.

Memory allocations, however, appear to be linear in growth with respect to the number of workers. We surmised that it is possible for space complexity to eventually reach O(H^2) given enough workers (enough being the total number of nodes in the graph), however it appears in practice that the growth is much closer to O(H).

However, having logarithmic performance gains with linear increases in memory usage is problematic. We will attempt to address these issues in *Foregoing Solution Construction* and *Distributed Computing*.

Comparison in Execution Time to Google Optimizations Tools

Google Optimization Tools (OR-Tools) is a fast and portable software suite for solving combinatorial optimization problems. [10] Specifically, we will be running comparisons against OR-Tools' constraint problem solving library, cpsolver.

	Language	Algorithm	Hardware
`search`	Go	Backtracking	Intel Core i7 2600k 16GB RAM
`cpsolver`	C++ with an interface to CPython	Constraint Logic Programming	Intel Core i7 2600k 16GB RAM

We will be using the NQueens program provided by Google in their tutorial page on cpsolver. [11] While this is not a perfectly one-to-one comparison (different languages and paradigms), it is still quite useful to compare our search statement against extant solutions that are agreed upon to be fast.

NQueens Size	`search` time (s)	`cpsolver` time (s)
1	0.000098	0
2	0.000084	0
3	0.000075	0
4	0.000074	0

5	0.000131	0
6	0.000311	0
7	0.001007	0
8	0.004274	0.002
9	0.024649	0.008
10	0.100438	0.031
11	0.478346	0.145
12	2.754791	0.75
13	15.693549	4.055
14	100.471620	23.045
15	652.413299	141.604

Single core performance comparison between the `search` statement and `cpsolver`. `cpsolver` provides timings at a maximum resolution of milliseconds.

cpsolver consistently outperforms the search statement in single core performance. This difference, however, is not in an entirely different complexity class. Given access to additional CPU cores, the search statement does catch up to the speed of cpsolver by the addition of the eighth core.

NQueens Size	`search` 8 workers time (s)	`cpsolver` single worker time (s)
1	0.000085	0
2	0.000076	0
3	0.000074	0
4	0.000075	0
5	0.000152	0
6	0.000234	0
7	0.000452	0

8	0.001367	0.002
9	0.005019	0.008
10	0.022851	0.031
11	0.109989	0.145
12	0.611703	0.75
13	3.429158	4.055
14	21.740712	23.045
15	139.937767609	141.604

Multi core performance comparison between the `search` statement and `cpsolver`. `cpsolver` provides timings at a maximum resolution of milliseconds.

It is unknown whether cpsolver can scale to hardware in a similar fashion to our search. However, from our benchmarks, it is fair to say that the single core performance of cpsolver is roughly equivalent to the eight core performance of search.

Given the circumstances (Go versus C++, backtracking versus constraint logic programming, natural performance with little optimization done, Google optimization research teams versus ASU graduate research) we are pleased with the performance of the search statement.

Foregoing Solution Construction

The solution data structure is the only data structure that is managed by the engine that passes through the engine:userland barrier. It is assumed that the user use holds an interest in this constructed solution is all cases. However, we have already visited a case where this is not necessarily true - the webcrawler.

```
seen := make(map[string]bool, 0)
search “asu.edu”; string; 8 {
children:
    c := make(chan Page, 0)
    hrefs := ExtractAllHrefs(node)
    seen[node] = true
    go func() {
       defer close(c)
       for _, href := range hrefs {
             if seen[href] {
                   continue
             }
             c <- href
       }
    }()
    return c
}
```

A webcrawler which makes no reference to the solution data structure.

The webcrawler makes no reference to the solution. It is maintaining another structure that may act as a proxy for the solution, the seen hashset. However all order is lost because the problem does not care about the order - it merely wants to visit all pages starting at asu.edu. However, the engine is still maintaining a slice of strings, recording all web pages that have been visited. Given a long running webcrawler the effects on memory consumption would be startling.

A proposed compiler optimization is to detect that the name solution is never referred to by the user code. Upon detection of such code, the engine can be swapped out for a version that does not maintain the any solution at all.

Due to time constraints this compiler optimization did not make it into the final implementation of the compiler, and is thus being left as a point of future research.

Distributed Computing

We have shown several critical, atomic, components to the search statement's story of scalability.

1. Graph searches may be concurrent.
2. The depth and width of a given instance of a search can be controlled by the user.
3. The user is capable of maintaining their own solution data structure.

4. In the future, the user may be able to hint to the compiler to not maintain an ongoing solution.

Our problem of scalability is as follows.

1. Given a reasonably bounded amount of time, a single compute node is unsatisfactory.
2. Even given an infinite amount of time, a single compute node will still eventually fail. The search engine requires a stack for internal management. Although this stack is allocated to the heap, even the heap is finite, and thus a given search must be finite.
3. Thus we must find a way to orchestrate a graph search across many compute nodes, where each compute node solves only a tractable portion of the graph.

Given that the only structure that is out of the user's control is the stack, it is possible for them to treat any sort of cycle detection or ongoing solution as remote storage. That is, although a given compute node may have access to dozens of gigabytes of RAM, a remote storage server may have access to many hundreds of gigabytes of RAM.

Consider the following worker for a distributed webcrawler. It has swapped out the use of a map for cycle detection with some unknown, remote, implementation. Additionally it is only considering a portion of the graph that goes as deep as the maxDepth with maxWidth being the maximum number of children a parent URL can have. And, finally, any URL that falls out of the maxDepth and maxWidth constraints are submitted to a remote supervisor which will dispatch that URL as the root for another instance of the same worker to consider. Upon completion of its work over its smaller, more tractable, subgraph the search engine core is dumped, the worker declares that it is ready for another portion of the graph. It waits until it receives such work from the supervisor and a new search is conducted.

```
type Page struct {
    URL   string
    Depth int
}

var maxWidth = 1000
var maxDepth = 1000

func receive() {
    for {
        // Receive work from supervisor.
        seed := acceptWork()
        crawl(seed)
    }
}

func crawl(seed string) {
    // Declares to the supervisor that this worker is busy.
    startWorker()
    // A remote, in memory, store used to share cycle detection amongst all workers.
    seen := NewRemoteCycleDetection(settings.RemoteCycleDection.Addr)
    // Submits new subgraphs for dispatch to available worker nodes.
    subsearch := NewRemoteWorkQueue(settings.RemoteWorkQueue.Addr)
    search Page{seed, 0}; Page; 8 {
    children:
        c := make(chan Page, 0)
        if node.Depth >= maxDepth {
                // This node is beyond this worker's max depth.
                // Submit it as work for another, available, worker.
                subsearch.Add(node)
                close(c)
                return c
        }
        hrefs := ExtractAll(node.URL)
        seen.Set(node) = true
        go func() {
                defer close(c)
                w := 0
                for _, href := range hrefs {
                        if seen.Get(href) {
                                continue
                        }
                        if w >= maxWidth {
                                // This node is beyond this worker's max width.
                                // Submit it as work for another, available, worker.
                                subsearch.Add(href)
                                continue
                        }
                        c <- Page{href, node.Depth + 1}
                        w++
                }
        }()
        return c
    }
    // Declare to the supervisor that this worker is available.
    stopWorker()
}
```

A worker in a distributed webcrawler.

Feature List

An idea that arose during the research was the feature of compiler provided memoization. The use of an externally provided memoization facility is not unprecedented (for example, the Python library functools provides the decorator lru_cache which memoizes results from the decorated function). Memoization is, of course, a trade off increasing space complexity in order to reduce time complexity. As such, we felt uncomfortable silently implementing memoization for all graph searches in general. We did not implement any such facility in the compiler, however it is worthwhile to look at what this interface may look like. The challenge in this is not so much the implementation as it is the presentation of the feature.

Already the search statement is nearing its limit for a small, concise, clean interface. However, if we can inject one more aggregate field then we can allow the user to readily modulate the behavior of the engine. Let us call this the *feature list*. The feature list may contain an unordered list of features or algorithms that the user wishes the engine to use.

Memoize	BFS with Cycle Detection
`[cache]`	`[cycle, bfs]`
`search FCG; Type {`	`search FCG; Type {`
`    ...`	`    ...`
`}`	`}`

An example of a memoized search and a BFS with cycle detection via use of the feature list.

Such a feature list would greatly expand the power and flexibility of the engine. However, the topic remains a point of future research.

WORKS CITED

1. The Rust Project Developers. 2010. The Rust Programming Language. Retrieved April 6, 2018 from https://doc.rust-lang.org/1.6.0/book/patterns.html

2. Wikipedia. 2018. Eight queens puzzle. Retrieved April 6, 2018 from https://en.wikipedia.org/wiki/Eight_queens_puzzle

3. Patrick Blackburn, Johan Bos, Kristina Striegnitz. 2012. Learn Prolog Now!. Retrieved April 3, 2018 from http://lpn.swi-prolog.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse6

4. Wikipedia. 2018. Backtracking. Retrieved March 31, 2018 from https://en.wikipedia.org/wiki/Backtracking

5. Goodrich and Tamassia; Cormen, Leiserson, Rivest, and Stein

6. Anton van Straaten. 2003. RE: What's so cool about Scheme? Retrieved March 31, 2018 from http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html

7. Jorge Aparicio. 2018. Rust by Example. Retrieved March 31, 2018 from https://rustbyexample.com/expression.html

8. Andrew Gerrand. 2013. The first Go program. Retrieved March 31, 2018 from https://blog.golang.org/first-go-program

9. Wikipedia. 2017. Cfront. Retrieved from https://www.wikipedia.org https://en.wikipedia.org/wiki/Cfront

10. Google Optimization Research. 2018. Retrieved April 3 from https://developers.google.com/optimization/

11. Google Optimization Research. 2017. Retrieved April 3 from https://developers.google.com/optimization/cp/queens

APPENDIX A

SEARCH ENGINE FULL CODE TEMPLATE

```
// This is the one piece of internals that the userland
// can potentially see due to scoping rules.
type __GraphNode struct {
    Active     bool
    ID          int
    Parent     int
}
// User CHILDREN declaration.
USER_children := func(node {UTYPE}, solution []{UTYPE}, gid *__GraphNode) <-chan {UTYPE} {
    return make(chan {UTYPE}, 0)
}
// User ACCEPT declaration.
USER_accept := func(node {UTYPE}, solution []{UTYPE}, gid *__GraphNode) bool {
    return false
}
// User REJECT declaration.
USER_reject := func(node {UTYPE}, solution []{UTYPE}, gid *__GraphNode) bool {
    return false
}
root := changeme
maxgoroutine := 1
// Parent:Children PODO meant for stack management.
type StackEntry struct {
    Parent   {UTYPE}
    Children <-chan {UTYPE}
}
lock := make(chan int, maxgoroutine)
wg := make(chan int, maxgoroutine)
ticket := make(chan int, maxgoroutine)
// You have to declare first since the function can fire off a goroutine of itself.
var engine func(solution []{UTYPE}, root {UTYPE}, gid *__GraphNode)
engine = func(solution []{UTYPE}, root {UTYPE}, gid *__GraphNode) {
    _children := USER_children(root, solution, gid)
    // Stack of Parent:Chidren pairs.
    stack := make([]StackEntry, 0)
    // Current candidate under consideration.
    var candidate {UTYPE}
    // Holds a StackEntry.
    var stackEntry StackEntry
    // Generic boolean variable
    var ok bool
    for {
         if candidate, ok = <-_children; !ok {
                 // This node has no further children.
                 if len(stack) == 0 {
                         // Algorithm termination. No further nodes in the stack.
                         break
                 }
                 // With no valid children left, we pop the latest node from the solution.
                 solution = solution[:len(solution)-1]
                 // Pop from the stack. Broken into two steps:
                 //    1. Get final element.
                 //    2. Resize the stack.
                 stackEntry = stack[len(stack)-1]
                 stack = stack[:len(stack)-1]
                 // Extract root and candidate fields from the StackEntry.
                 root = stackEntry.Parent
                 _children = stackEntry.Children
                 continue
         }
         // Ask the user if we should reject this candidate.
         _reject := USER_reject(candidate, solution, gid)
         if _reject {
                 // Rejected candidate.
                 continue
         }
         // Append the candidate to the solution.
         solution = append(solution, candidate)
```

```
		// Ask the user if we should accept this solution.
		_accept := USER_accept(candidate, solution, gid)
		if _accept {
			// Accepted solution.
			// Pop from the solution thus far and continue on with the next child.
			solution = solution[:len(solution)-1]
			continue
		}
		select {
			case lock <- 1:
				wg <- 1
				s := make([]{UTYPE}, len(solution))
				copy(s, solution)
				go engine(s, candidate, &__GraphNode{Active: true,
						ID: <-ticket, Parent: gid.ID})
				// pretend we didn't see this
				solution = solution[:len(solution)-1]
				continue
			default:
		}
		// Push the current root to the stack.
		stack = append(stack, StackEntry{root, _children})
		// Make the candidate the new root.
		root = candidate
		// Get the new root's children channel.
		_children = USER_children(root, solution, gid)
	}
	<- lock
	wg <- -1
	gid.Active = false
}
shutdown := make(chan struct{}, 0)
go func() {
	// Goroutine ticketing system.
	id := 0
	for {
		select {
		case ticket <- id:
			id++
		case <-shutdown:
			close(ticket)
			return
		}
	}
}()
lock <- 1
wg <- 1
go engine(make([]{UTYPE}, 0), root, &__GraphNode{Active: true, ID: <-ticket, Parent: 0})
count := 0
for c := range wg {
	count += c
	if count == 0 {
		break
	}
}
close(shutdown)
close(wg)
close(lock)
```

APPENDIX B

NQUEENS BENCHMARK DATA

Table 1: Search Statement Benchmarks (Intel i7 2600k, 16GB)

Value 1	**Value 2**	**Value 3**	**Value 4**
NQueensSize	*Cores*	*Time(s)*	*MemoryAllocations*
1	1	0.000098	13
1	2	0.000096	12
1	3	0.000093	12
1	4	0.000090	11
1	5	0.000088	12
1	6	0.000085	12
1	7	0.000086	11
1	8	0.000085	12
2	1	0.000084	15
2	2	0.000084	17
2	3	0.000081	18
2	4	0.000078	17
2	5	0.000080	17
2	6	0.000077	17
2	7	0.000076	17
2	8	0.000076	17
3	1	0.000075	19
3	2	0.000074	26
3	3	0.000073	29
3	4	0.000074	28
3	5	0.000075	29
3	6	0.000074	28
3	7	0.000073	29
3	8	0.000074	29
4	1	0.000074	31
4	2	0.000075	43
4	3	0.000074	51
4	4	0.000075	56
4	5	0.000074	58
4	6	0.000073	59
4	7	0.000074	59
4	8	0.000075	60
5	1	0.000131	62
5	2	0.000132	86
5	3	0.000138	109
5	4	0.000145	126
5	5	0.000148	142
5	6	0.000152	156
5	7	0.000149	157
5	8	0.000152	157
6	1	0.000311	168
6	2	0.000287	217
6	3	0.000269	246
6	4	0.000262	289
6	5	0.000257	321
6	6	0.000259	355
6	7	0.000248	383
6	8	0.000234	396

Table 2: Search Statement Benchmarks (Intel i7 2600k, 16GB)

Value 1	**Value 2**	**Value 3**	**Value 4**
$NQueensSize$	$Cores$	$Time(s)$	$MemoryAllocations$
7	1	0.001007	531
7	2	0.000814	658
7	3	0.000656	714
7	4	0.000625	819
7	5	0.000563	894
7	6	0.000515	984
7	7	0.000475	1048
7	8	0.000452	1092
8	1	0.004274	1986
8	2	0.002975	2291
8	3	0.002398	2525
8	4	0.002046	2804
8	5	0.001855	3096
8	6	0.001634	3297
8	7	0.001465	3489
8	8	0.001367	3608
9	1	0.024649	8065
9	2	0.012764	9104
9	3	0.009193	9861
9	4	0.007937	10975
9	5	0.006887	11885
9	6	0.006004	12499
9	7	0.005432	13303
9	8	0.005019	13786
10	1	0.100438	34853
10	2	0.059459	38762
10	3	0.043260	41919
10	4	0.036333	45474
10	5	0.031644	49152
10	6	0.027544	51464
10	7	0.024762	53735
10	8	0.022851	55413
11	1	0.478346	164321
11	2	0.291113	182030
11	3	0.204158	191878
11	4	0.173678	207941
11	5	0.151078	223552
11	6	0.132125	233443
11	7	0.119249	243708
11	8	0.109989	250405
12	1	2.754791	842409
12	2	1.595484	913660
12	3	1.141065	975535
12	4	0.955877	1041931
12	5	0.824667	1101564
12	6	0.727172	1157337
12	7	0.658361	1202046
12	8	0.611703	12413891

Table 3: Search Statement Benchmarks (Intel i7 2600k, 16GB)

Value 1	**Value 2**	**Value 3**	**Value 4**
NQueensSize	*Cores*	*Time(s)*	*MemoryAllocations*
13	1	15.693549	4602944
13	2	8.995142	4971970
13	3	6.385116	5269324
13	4	5.340870	5595718
13	5	4.618225	5898360
13	6	4.078673	6173339
13	7	3.688601	6374454
13	8	3.429158	6574777
14	1	100.471620	27006112
14	2	56.033450	28827162
14	3	39.852565	30374654
14	4	33.613693	32351672
14	5	29.072379	33931816
14	6	25.740943	35487338
14	7	23.342292	36522323
14	8	21.740712	37545358
15	1	652.413299	168919148
15	2	359.393896	179303271
15	3	257.803043	189516332
15	4	215.340724	199520814
15	5	187.121518	209514463
15	6	165.471322	217867713
15	7	150.155104	223703983
15	8	139.937767609	229658340