Optimization Apps: Combinatorial Optimization, Linear, Integer, and Nonlinear Programming Algorithms (COLINA) Grande

J. MacGregor Smith
Mechanical and Industrial Engineering University of Massachusetts Amherst, MA 01002 jmsmith@ecs.umass.edu

This book is a monograph about the development of optimization Apps for personal computing. Many different optimization problems are explained and the use of MIT's App Inventor 2 is utilized for their solution. Many of the apps are self contained and the problem formulation, algorithm, and expected results are shown. Use of App Inventor 2 for solving larger problem instances is also discussed which involves sending the apps to the Network-Enabled Optimization System (NEOS) server at the University of Wisconsin. Other possible extensions of the apps to other computing languages and environments are also possible. The idea of this volume is to distill the principles one needs to create optimization apps for whatever computing environment comes along. Another motivating principle is to engage the student in a modelling exercise to recognize a system problem that could be improved and creative ways to improve it. Finally, learning programming with App Inventor adds to one's skill set.

Key words: Apps, Optimization, Personal Computing

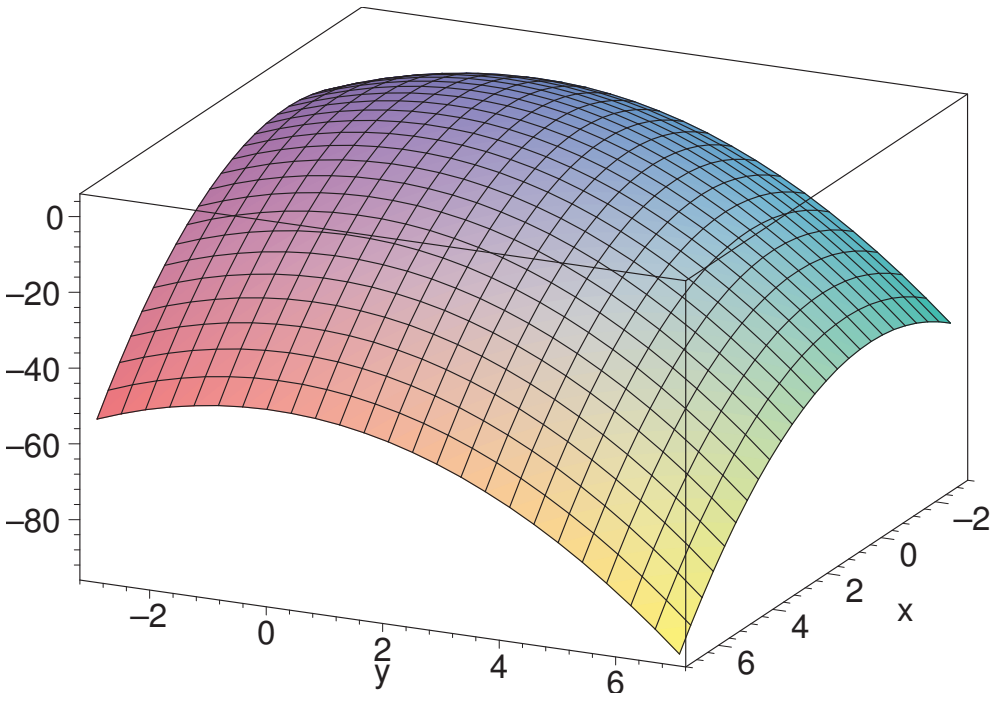


Figure 1. COLINA Grande Optimization

Contents

1. Introduction

Personal mobile computing is the one of key revolutions which has already set upon us. Apps on phones and tablets are commonplace and becoming more important for today's survival. We desire to create optimization apps which are not just descriptive of a situation but prescriptive so that one can make things better. These are more challenging than simple descriptive apps, but on the other hand, more rewarding in the long run.

We would like for students to be creative consumers not just consumers of the new technology. How can we benefit and realize this process and how can we teach our students at the university level how they can contribute to making things and making them better, no matter what engineering, science, or business discipline they originate from.

On a personal note, the origins of my interest in apps came one day from reading the Business section of the Sunday New York Times in 2010 about a programming class at Stanford where students were learning how to program a phone app on the Apple phone which they could market. Why cannot we do this at UMass? Well, one thing led to another and I found out about App Inventor which was just starting at Google. This was the first instance of App Inventor classic version #1. In the Fall of 2011, I started teaching about App Inventor to my Mie 379: *Introduction to Operations Research* class and said I wanted to have the students carry out an App Inventor optimization term project. I felt that creating an optimization app would be challenging but give our students a unique slant on the app market. I think this intuition was correct. It was crazy ambitious since the App Inventor programming language was in its initial developing stages. The thing that grabbed my attention was the ability to design the user interface for the phone with all the buttons, labels, text and diagrams. The students and I suffered through the first class, but, then, this was to be expected. Since then, and after many classes, the materials in this book reflect the learning and development which has occurred over time. App Inventor has matured and been very successful in teaching programming skills from elementary school to higher education and beyond.

1.1. Why this Volume

Why are Operations Research and Computer Science relevant and important? Operations Research (OR) is relevant and important because most all optimization problems require some theoretical and applied mathematics understanding as their foundation. The formulation of optimization problems is founded through OR concepts and techniques: Combinatorial Optimization, Linear Programming, Integer and Nonlinear Programming (COLIN). Computer Science (CS) is relevant and important because algorithms (A) underly the solution of all optimization problems. App Inventor and AMPL are the major algorithmic vehicles studied in this course. Thus, in summary we have COLINA which means "small hilltop" in Spanish and it is Grande! Figure 1 illustrates a small hilltop optimization problem in the variables x and y.

For whom is this book written? University level students and professors, mainly at the upper division level or higher with some background in calculus, linear algebra, and related mathematical skills. These skills are viewed as sufficient but not necessary for developing the Apps for Optimization, *i.e.* OptApps or AppOpts.

How is this book organized? Basically, each chapter provides the theoretical background and the general methodology for the concepts of optimization according to COLINA, then there are example applications of the Apps with App Inventor and AMPL software. The COLINA categories are representative of the major areas of interest in optimization which I have come across in my teaching experience. Linkages to the University of Wisconsin NEOS server for their solution can also occur. Some of the APPs are stand alone which is ultimately flexible, but others require more computing firepower.

1.2. Wicked Problems

We need to realize that most real world design and planning problems that we wish to optimize are extremely challenging and difficult, in fact, they are often viewed as Wicked Problems (WPs). However, we need to approach these problems with algorithms and a desire to solve them as best as possible seeking the crest of the local optimal solutions. The notion of WPs was coined by Horst Rittel [3]. Wicked problems have the following challenging properties or characteristics:

- $\mathcal{P}_1$ No definitive problem formulation.
- $\mathcal{P}_2$ No exhaustive list of permissable operations.
- $\mathcal{P}_3$ No stopping rule.
- $\mathcal{P}_4$ No **single** criterion for correctness.
- $\mathcal{P}_5$ Many **alternative solutions**.
- $\mathcal{P}_6$ Every WP is symptomatic of another WP.
- $\mathcal{P}_7$ No immediate or ultimate test of a solution.
- $\mathcal{P}_8$ Every WP is a one-shot operation.
- $\mathcal{P}_9$ Every WP is essentially unique.
- $\mathcal{P}_{10}$ We are morally responsible for a solution.

The set of properties of a WP indicate its richness and difficulty. $\mathcal{P}_1$*: No definitive problem formulation* implies that a proper problem formulation indicates the actual solution, however, there is no single way to define the problem since it depends upon your worldview. Your worldview not only helps you understand the problem, but it colors your perspective. C. West Churchman, the famous systems scientist, was instrumental in gathering people together so that many worldviews would be present in resolving system planning problems, not just one. In this book, we have gathered together a number of different problems and applications to illustrate the variety of approaches in problem solving.

$\mathcal{P}_2$*: No exhaustive list of permissable operations* means that there is no exhaustive set of steps to guarantee a solution. We could read all the books on a subject, ask all the important people affected by the problem, observe the situation, and so on. There are many alternative methods and ways to approach a problem concomitant with the different worldviews. A person's worldview is crucial to the approach for solving the problem.

$\mathcal{P}_3$ *No stopping rule* implies that you cannot stop the solution process because you can always do better. This is most frustrating for decision makers since the word *decidere* comes from the Latin "to cut off". Normally, people give up because they have exhausted all their resources.

$\mathcal{P}_4$ *No* **single** *criterion for correctness* indicates that the problems are multi-objective in nature involving often intricate complicated tradeoffs. There can be many alternative solutions since the problem is multi-objective and has a complex noninferior set of solutions[1]. The noninferior set can have an infinite number of solution alternatives, not just a single one.

$\mathcal{P}_5$ *Many* **alternative solutions** is concomitant with the multi-objective nature of the problem, there are many alternative solutions, not one best solution.

$\mathcal{P}_6$ *Every WP is symptomatic of another WP* means that the WPs are interdependent and nested together and not separable. Normally, we like to decompose a problem into separate pieces, but WPs are anathema to this strategy.

$\mathcal{P}_7$ *No immediate or ultimate test of a solution* implies that there is no immediate or ultimate way to guarantee a solution's correctness. There is no simple simulation model to test the robustness of the solution. The fact that there are many alternative solutions and no single criterion for success mean that we have a no guarantees.

$\mathcal{P}_8$ *Every WP is a one-shot operation* usually means that the resources needed to solve a WP imply that you only have one chance to solve it without major consequences for adjusting your solution. Normally, the expenses of tearing down and starting over far outweigh that strategy.

[1]Noninferior for a multi-objective problem means that there exists no other feasible solution that will yield an improvement in one objective without causing a degradation in at least one other objective. [1]

$\mathcal{P}_9$ *Every WP is essentially unique* generally means that each WP is unique in time and space. We cannot learn by solving WPs over and over, we must start anew again. This is a most frustrating property since we surmise from past experience that our learning about problem solving provides a straightforward framework and underpinning for solving new problems. But this is just magical thinking, there are no experts.

$\mathcal{P}_{10}$ *We are morally responsible for a solution* implies that we are ethically responsible for our actions and we have to be morally responsible in our decision-making. In engineering, architecture, law, and most professional disciplines, we have an obligation to society to provide the best advice possible.

While real world problems are WPs, we still have to work through them. This is why we need computing help and algorithms in order to deal with WPs.

As an interesting example of a WP, let's examine the process of selecting a golf club to hit a golf shot. This is seemingly a trivial problem to the non-golfer ("its only a game"), but in reality, for the average golfer this problem is most complex. Even for the pros, this is challenging, and why they are so inconsistent. Why is this the case? It really stems from the fact that each time the ball lands, it ends up in a different lie situation, so it makes the problem unique. The ball can be in a wet sand trap; on the downside or upside of a hill; there is moisture or dirt on the ball; the ball lies on bare ground; there is a twenty mile and hour wind blowing in the golfer's face; and so on. To represent the actual situation requires critical information and there is a great deal of *uncertainty and variability* as to how to assess the situation.

- What is the true distance to the pin when the green is situated 10 feet above your position on the course?
- In match play, should you play safe or take a risky shot, since your partner's situation also becomes a factor to consider?
- In the wet rough, should you use an iron or a fairway wood because the fairway wood will be likely be impeded by the moisture?
- Should one play over or under the trees?
- Should you bump and run or fly it to the pin from the fairway?

There is only one chance for success. We want to do our best for the average golfer. To do the job right, how should we design a phone app for helping the average golfer. Thus, providing an optimization app for advice on how to hit their shots to the average golfer is very challenging for the app designer.

Computer Science has developed a set of problem categories that are useful in designing algorithms for these problem solutions. It is a very useful characterization because if one knows the category of problem, then one can assess the difficulty of the problem solution and an appropriate algorithm.

The problem categories are based on a fundamental abstract model of computation called the Turing Machine (TM) model of computation, or Deterministic Turing Machine (DTM) for short. A DTM is a model representing most practical, everyday computers. Another useful abstract comparison model is called a Nondeterministic Turing Machine (NDTM) for short. A NDTM does not really exist, it is fictitious, but it is helpful in the problem classifications. Briefly, here are the complexity classes:

- $\mathcal{P}$, The class of problems which are solvable by a DTM in polynomial time.
- $\mathcal{NP}$, The class of problems which are solvable in polynomial time on a NDTM.
- $\mathcal{NP}$-Hard, The class of problems at lest as hard as the hardest problems in $\mathcal{NP}$.
- $\mathcal{NP}$-Complete, The class of problem in $\mathcal{NP}$ and the intersection of $\mathcal{NP}$-Hard which are transformable in polynomial time to the *3-Satisfiability* problem.
- *Undecidable* are those problems where it is impossible to construct an algorithm (TM) that always leads to a correct yes-or-no answer. In effect, the Undecidable problems are almost equivalent to the Wicked Problems described earlier, but WPs remain worse than the Undecidable ones.

Figure 2 illustrates the relationships between the problems of Computer Science and those problems we call Wicked Problems. Most of the problems we examine in this book will be from the class $\mathcal{P}$ simply because of the phone or tablet's limited computational power. Sometimes an $\mathcal{NP}$-Hard problem can be solved optimally if the size of the problem is small. That is why the link to the NEOS server is provided, but even still, Wicked Problems are truly difficult. The key issue facing us is: *Can we solve WPs with systematic procedures?*

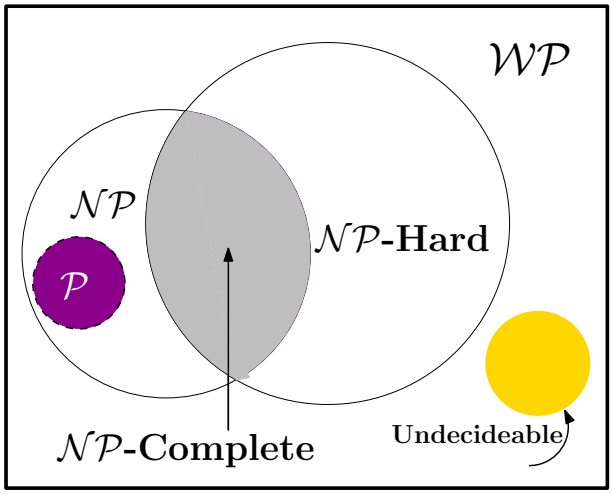


Figure 2. Relationship between Computing and Wicked Problems

1.3. Systematic Procedures and Algorithms

In this course and book, we will examine COLINA type problems with basically nine systematic procedures and algorithms for their solution. The range of algorithms is appropriate depending on the particular type of problem. Many problems will be solved by combining together some of the strategies. The idea is to widen our approaches for solving problems rather than to rely only on a singular technique such as Linear Programming (LP). We give a brief definition of the different techniques.

- **Sorting and Searching:** Sorting and search methods are very basic and should be included first as they are classical procedures that are part and parcel of an algorithms library and can be very useful in different applications.
- **Greedy Solutions**: Solving a problem by carefully selecting components of the problem in a greedy fashion and putting them together for a solution of the entire ensemble is very sensible, but is normally an heuristic approach.
- **Divide-and-Conquer** (*a.k.a* **Branch & Bound**): Partitioning a problem into different sets and recursively constructing a solution for integrating the elements within the partitions.
- **Backtracking**: Essentially a process for enumerating all the combinatorial solutions of a problem. Normally, all the different solutions are represented as a tree and the solutions are developed by moving down the branches of the tree and backtracking to the top of the tree and searching for a new path again until all solution paths have been enumerated.
- **Linear Equations**: Developing a set of linear equations for a problem and solving them by Gaussian methods, Gauss-Jordan, or Chelosky decomposition.
- **Linear Programming (LP)**: Essentially setting up a linear objective function and constraints for a problem and using standard methods such as the simplex or dual-simplex method for their solution. This is one of the basic methods taught in the Mie 379 course.

- **Dynamic Programming**: Use of Bellman's equations and the concepts of state space and stages in a recursive approach to enumerate solutions to a problem. Certain problem types which have a recursive nature to them lend themselves to this technique.
- **Nonlinear Equations and Programming (NLP)**: Use of the calculus to generate derivatives or Lagrangian techniques to solve a set of nonlinear equations.
- **Computational Geometry**: Use of convex hulls, geometric searching, Voronoi diagrams or Delaunay triangulations to solve a geometric optimization problem. Location optimization problems are very amenable to this approach.

Within the book and in the various applications discussion, we shall illustrate the strategies and algorithms rather than as a separate abstract chapter on the techniques themselves. It is felt that the strategies are best seen within the application problems.

There are many caveats to mention here since App Inventor itself is not a sophisticated programming language. For optimization, it has no libraries, so things like sophisticated data structures (*e.g. matrices*) or sorting routines do not exist, one has to build them from scratch. This is problematic but not insurmountable, however, one has to be realistic. It has no software graphics library itself and graphics must be imported. The programming blocks are very visual and nicely designed which is a real plus. Also, as has been mentioned, when more computing power is needed there is always the NEOS server. However, it is best if the app can be self-contained. So the problem formulation of the app idea is really crucial.

1.4. Plan of the Book

The plan of the book is to introduce each app in a concise framework as best as possible with the following organizational structure:

- Introduction
- Problem
- Mathematical Model
- Algorithm
- Solution App
- App Demonstration
- Evaluation (Benefits and Costs)

App Inventor basically has three main steps. First, one must design the layout of the screen with all the buttons, divisions of space, and labels. This is the fun part and the selection of the items to be included on the phone is pretty rich. The second part deals with the programming blocks. This is usually the hard part, which also includes any debugging. The third part is the demonstration of the app with the phone emulator or actual phone or tablet. This also can be fun.

Since the apps are designed to be optimization tools, the structure of the apps is somewhat unique because we are basically developing mathematical models for solving optimization problems. Within App Inventor, the strategy I normally follow in programming the blocks for an optimization problem is to:

1. Identify the decision variables and parameters.
2. Initialize their values.
3. Set up any list data structures.
4. Identify the objective function and any constraints (equalities and inequalities).
5. Develop procedures to process the variables according to the desired strategy for solving the problem.
6. Set up any restarting procedures to re-set the variable values and iteratively test and re-run the app.

The last step is important since one wants to evaluate the app process implementation. Let's demonstrate some of the apps according to our COLINA framework.

2. Combinatorial Optimization (CO)

Combinatorics is generally concerned with the arrangement, grouping, ordering, or selection of a discrete set of objects usually finite in number [2]. Combinatorial Optimization (CO) is concerned with finding the "best" arrangement, grouping or ordering of a set of discrete objects. Combinatorial Optimization problems are a sub-class of OR and CS problems for which many applications abound. There are many different types of CO problems which are normally framed in terms of a graph context $G(V, E)$ where G is the graph with a finite set of nodes V and edges E. There are sets of alternatives and we usually try to compare the objects in the sets until one of them emerges as the best solution. Occasionally, we enumerate all the objects in the sets to find the best alternative. A profit/cost function is usually supplied for comparing the different alternatives. For instance, Figure 3 illustrates a graph with a finite number of nodes and edges and we might wish to find the shortest path between node S and node T where the edges have a distance, cost or a reliability value.

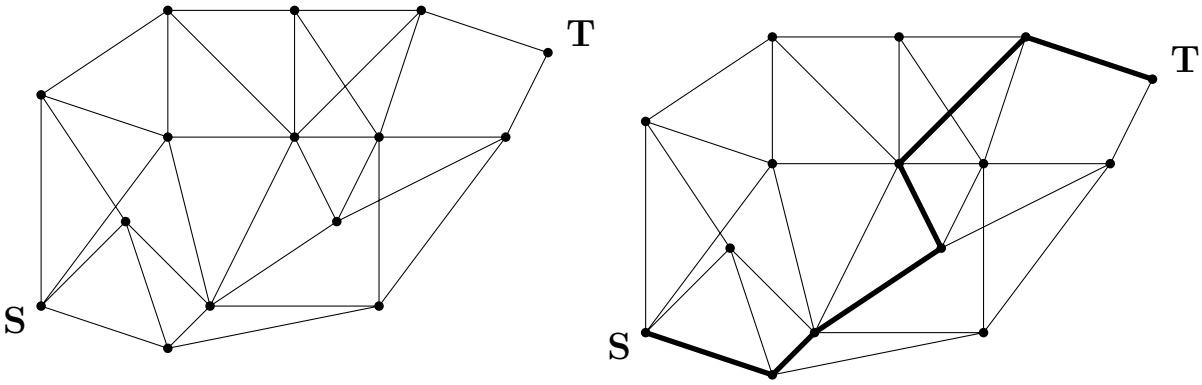


Figure 3. Example Graph

One particularly important CO problem is the Travelling Salesman Problem (TSP) discussed next.

2.1. Stowe Cycle TSP

The TSP is one of the most well-known and most important CO problems. Many people recognize its significance and universality. There are two major parts of the problem. The first is to find a feasible tour through the set of nodes V also known as a Hamilton cycle and the second problem is to find the optimal Hamilton cycle of all possible ones. The first problem is $\mathcal{N}P$-Complete, while the second is $\mathcal{N}P$-Hard.

2.1.1. Introduction Let's say that we run a bicycle touring club in Stowe, Vermont and we wish to plan a tour for our group members in the Stowe, Vermont area.

There are a number of scenic trails in and around the town and you wish to plan a small tour of the town for your friends that is both challenging and interesting. Figure 4 represents a map of Stowe, Vermont. The four principal stops in our tour are:

SV: Stowe Village, the main area of the town.
SC: The old school house on School street leading away from the village.
EB: Emily's Bridge which is an historic covered wooden bridge.
LV: The Lower Village which is a commercial area along Route 100.

What is of importance here is not necessarily the stops but the edges/arcs connecting them which reflect the beautiful views and landscape and the ever changing levels in elevation that provide the challenge for the bike riders.

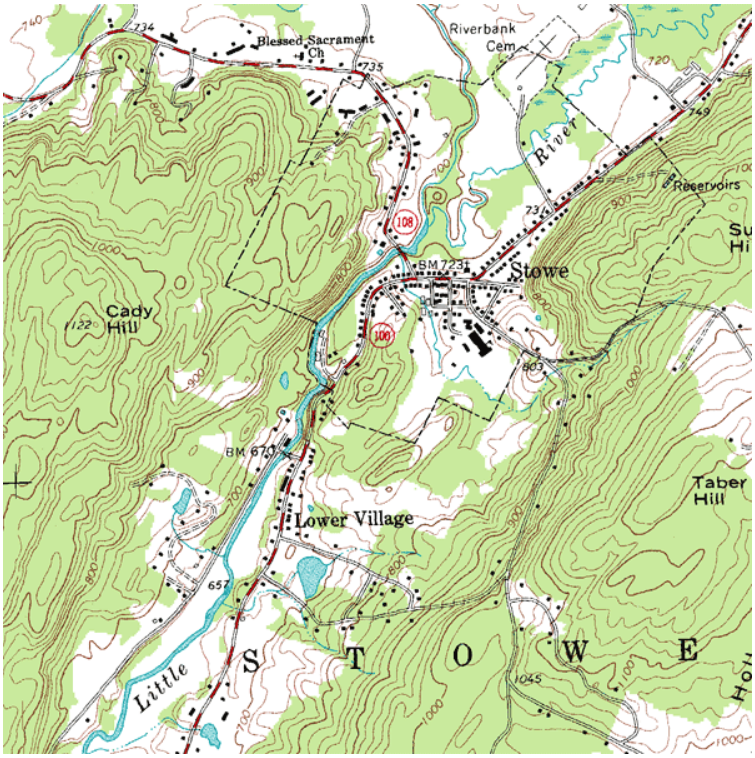


Figure 4. USGS Map of Stowe, Vermont

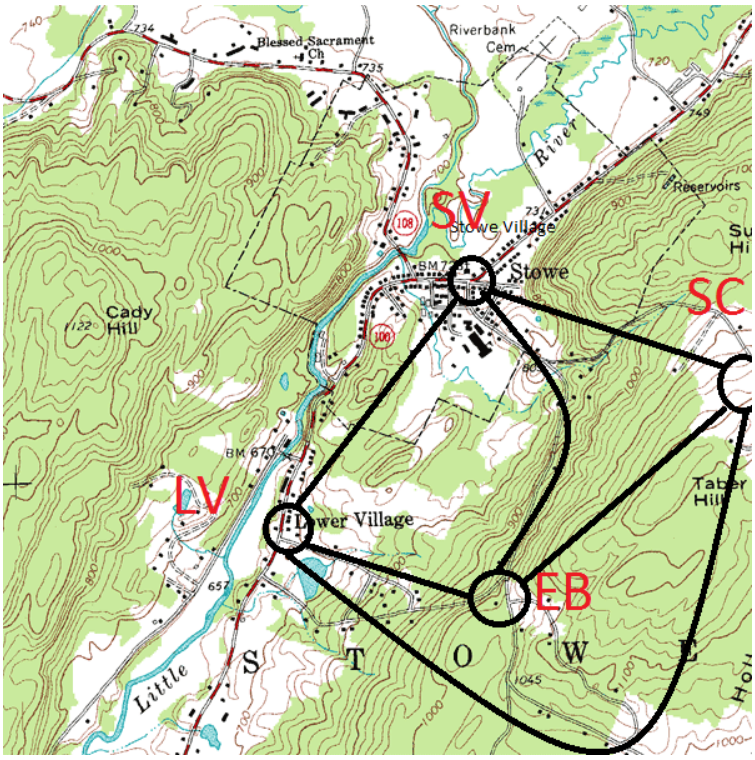


Figure 5. G(V,E) Tour

2.1.2. Problem In one sense, if we consider the arcs connecting the points of interest as the road distances between them, we seek the most interesting tour between them. Since there can be n-possible tour sites, it can be shown that there are $(n-1)!$ possible tours. We need a way, an algorithm, to find the "best" route.

If we think about the problem **carefully**, we see that because the tour is a cycle (no pun intended), it does not matter which tour stop we start with, it must always go through the starting stop, so we can eliminate one stop from the possible combinations, then only have to examining the remaining $(n-1)$ permutations. Figure 6 illustrates the conceptual ideas for a TSP solution.

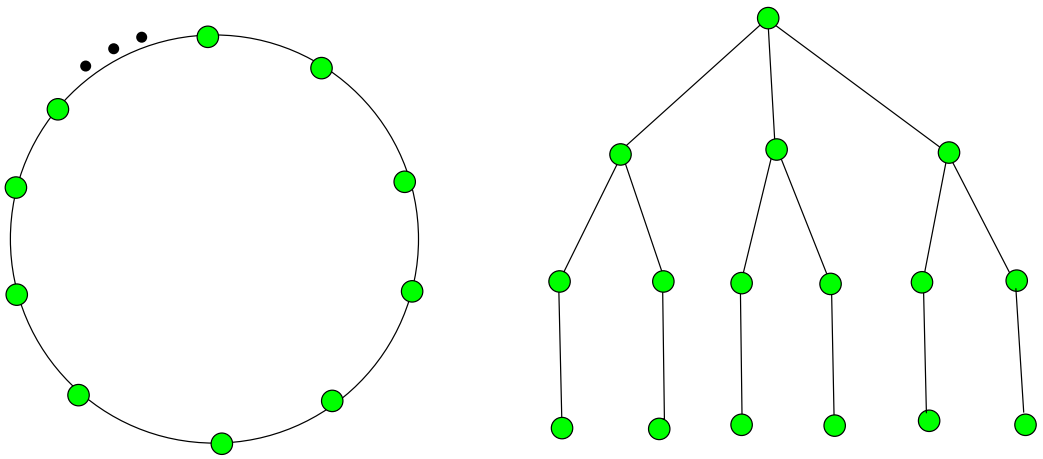


Figure 6. a)Cyclic Nature of Problem (left) b) Enumeration Tree (right)

Given this situation, we see that because there are only four stops on the tour, we could exhaustively enumerate all possible tours $(n-1) = 3! = 6$ and pick the best one. In general, this is **not** a good idea if n is larger $n \geq 10 \approx ((n-1) = 9!)$, but is ok, for smaller values and we can get an optimal solution. Enumeration of the tours makes the problem easily fit in the $\mathcal{N}P$-Complete class of problems. So we are following a backtracking strategy as discussed earlier.

Given a **distance matrix** connecting the tour stops, we can then evaluate the different possible tours. The values of the distance matrix are based on a scale from $1 \rightarrow 10$ where 10 is the *most interesting and the least stress*, while 1 is the *most stress and least interesting*. These objectives actually are conflicting in nature so even our simple bicycle planning problem is a WP. What other types of objective functions could be used? Distance, costs, time, profits and quality come to mind.

Below is the preference matrix between the stops on the tour. Whose preferences are these and how do we measure them? We want to **maximize** our preference rather than minimize the distance between stops on the tour. Minimizing distance is normally the way people solve this TSP problem. Preferences are difficult to quantify in general. Do you prefer coffee with milk or coffee without? How would you quantify the difference or would you be indifferent? Anyway, if we place the preferences on a scale

from *(lowest) 1-10 (highest)* then it becomes a reasonable vehicle to compute the tour transitions.

$$\begin{bmatrix} & 1(villlage(lv)) & 2(school(sc)) & 3(bridge(eb)) & 4(lower(lv)) \\ 1(sv) & \mathbf{0} & \mathbf{6} & \mathbf{8} & \mathbf{3} \\ 2(sc) & \mathbf{2} & \mathbf{0} & \mathbf{5} & \mathbf{9} \\ 3(eb) & \mathbf{6} & \mathbf{3} & \mathbf{0} & \mathbf{6} \\ 4(lv) & \mathbf{4} & \mathbf{4} & \mathbf{5} & \mathbf{0} \end{bmatrix}$$

Below is the list of possible tours:

$$\begin{array}{ll} \text{a: } 1\to 2\to 3\to 4\to 1 & \sum = 21 \\ \text{b: } 1\to 2\to 4\to 3\to 1 & \sum = 26 \\ \text{c: } 1\to 3\to 2\to 4\to 1 & \sum = 24 \\ \text{d: } 1\to 3\to 4\to 2\to 1 & \sum = 20 \\ \text{e: } 1\to 4\to 2\to 3\to 1 & \sum = 18 \\ \text{f: } 1\to 4\to 3\to 2\to 1 & \sum = 13 \end{array}$$

2.1.3. Mathematical Model We want to achieve the optimal solution in a general situation! One mathematical formulation of the problem is given below and it shows that if you could create this formulation one could solve it with a backtrack type algorithm, called branch-and-bound.

M_a Mathematical Sets: (i,j) tour stops

P Parameters: n:= number of tour stops; p_{ij}:= cost/benefit or in our case the preferences to travel from stop i to stop j

D Decision variables: $x_{ij} = 1$ if we travel from stop i to stop j, and $x_{ij} = 0$ otherwise.

O Objective function:

$$\text{Maximize } Z = \sum_i \sum_j p_{ij} x_{ij}$$

C Constraints:

$$\sum_i x_{ij} = 1 \text{ for all } j \tag{1}$$

$$\sum_j x_{ij} = 1 \text{ for all } i \tag{2}$$

$$\text{no subtours allowed} \tag{3}$$

$$x_{ij} \in \{0,1\} \text{for all } i \text{ andl } j \ : j > i \tag{4}$$

The general mathematical model objective function looks to sum the preferences for all the possible tours, subject to the fact that each tour enters each node exactly once Equation (1), and visits all nodes exactly once, Equation (2). The difficulty with the problem is to avoid the subtours that can occur unless one is careful in the enumeration process.

2.1.4. Algorithm The designer screen for AI2 is the primary vehicle for setting up the app representation. App Inventor 2 (AI2) as was discussed is basically comprised of three parts:

- **I. Designer Screen:** Input of images and parameter set-ups.
- **II. Visual Blocks Programming:** Logic, control, and procedural programming constructs
- **III. Emulation or Phone App:** Feedback and Demonstration of App

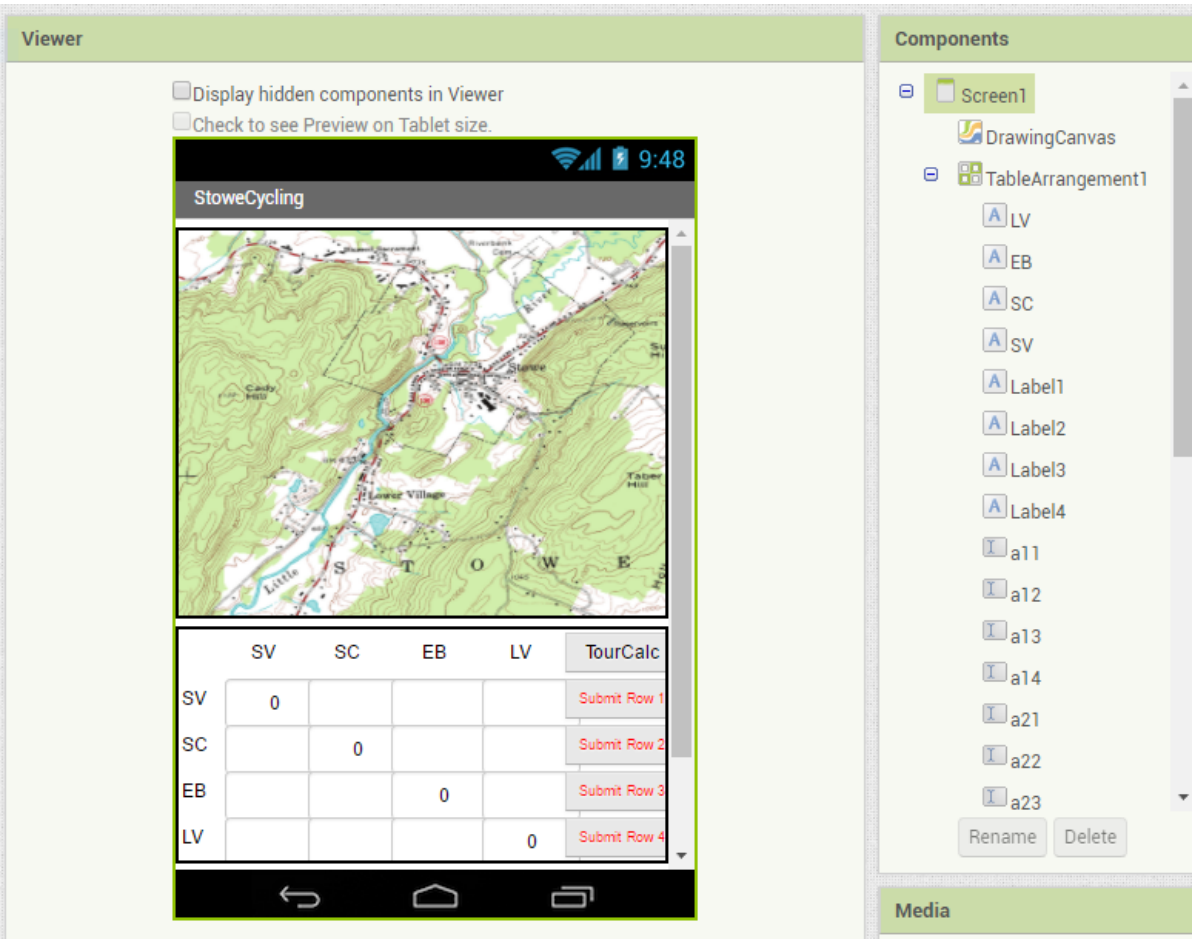


Figure 7. Stowe Cycle App Inventor Screen

Figure 7 illustrates the designer screen for our app where all the parameters are established, along with the buttons, distance matrix, images and screen inputs.

This TSP actually turns out to be a difficult combinatorial optimization to solve in general for a large number n of stops. Since the Stowe Cycle problem is very small, we can solve it with our enumeration algorithm implemented in AI2. Figure 8 is the set of blocks for the algorithm for the Stowe Cycle problem in AI2. It was pretty straightforward to program since only addition was required for the alternative routes.

2.1.5. Demonstration Here in Figure 9, is a picture of the actual optimization app. On the left is the input screen and on the right is the tour calculation with $Z = 26$ at the top of the screen and the optimal tour $1-2-4-3-1$ at the bottom of the screen. Finally, after inputting the sample data and running the backtrack algorithm through the phone app, we have as might be expected:

$$1 \to 2 \to 4 \to 3 \to 1 \qquad \sum = 26 \tag{5}$$

Of course, for an alternative preference objective function we would rely on the phone app for its solution after we input our data.

2.1.6. Evaluation The app works for a very limited number of tour stops but it is designed to show the general structure of the backtracking strategy for combinatorial optimization problems. It is nice and compact for inputting the preference structure and the output is relatively fast for the problem.

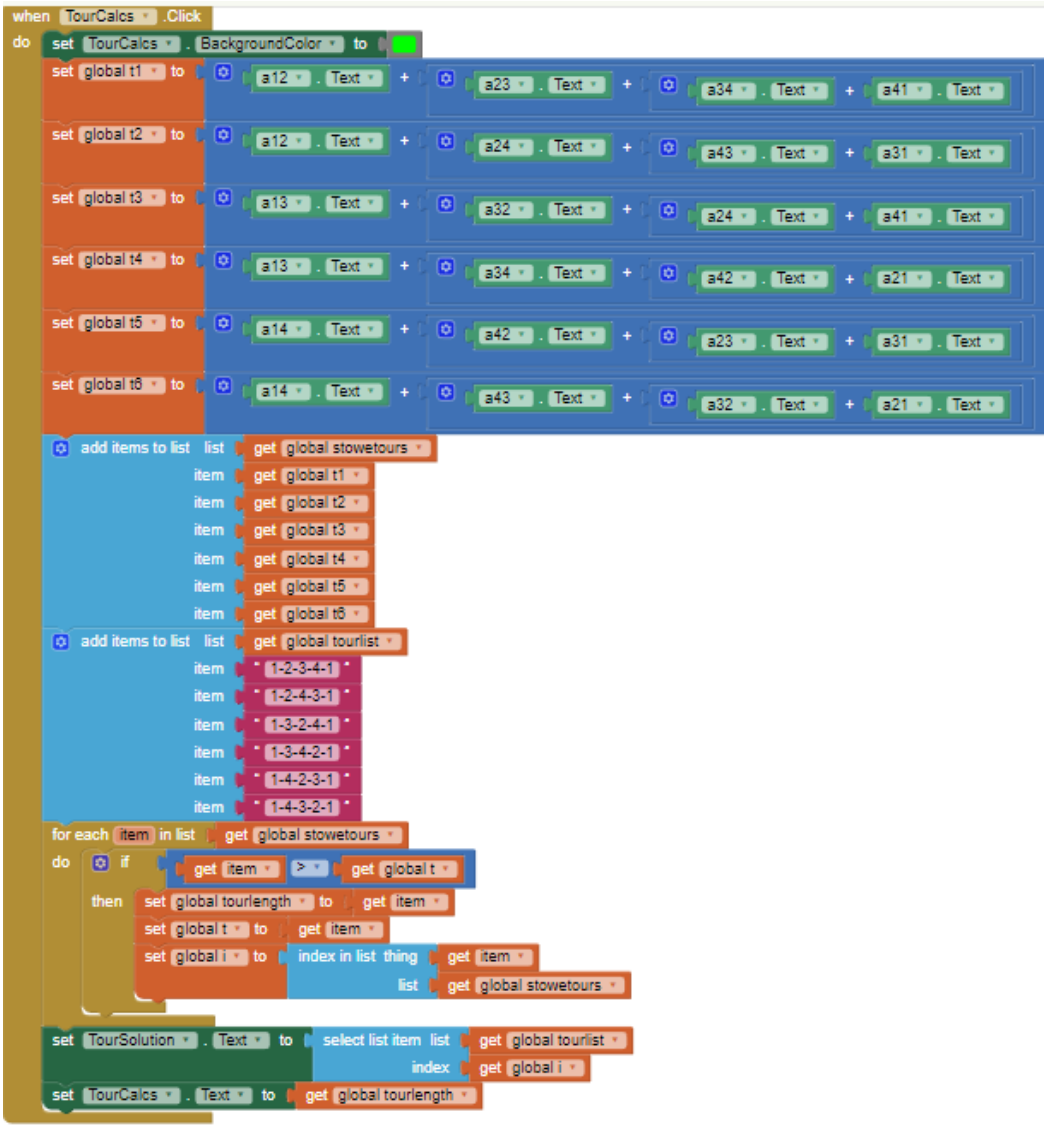


Figure 8. Complete AI2 Blocks Stowe Cycle Algorithm Solution

It is limited to a small number of stops. Large problem instances where n is allowed to range have to wait till §6 where we can employ the NEOS server to solve larger problem instances. This larger TSP problem is discussed in the last section of the book.

Now let's examine a different problem concerned with warehouse management. It is somewhat related to the TSP problem, but can be treated from many different viewpoints, as can many of the WPs.

2.2. Warehouse Quick Pick

A very practical industrial engineering problem is concerned with an order picking system in a warehouse. As part of a supply/chain operation, order picking can be defined as the systematic activity of selecting a small number of goods from a warehouse system in order to satisfy a number of customer orders. We could set it up as a TSP, but we want to examine a different approach and show that even if we have a technique for solving a problem, formulating it from a different viewpoint will generate a different solution, $\mathcal{P}_1$.

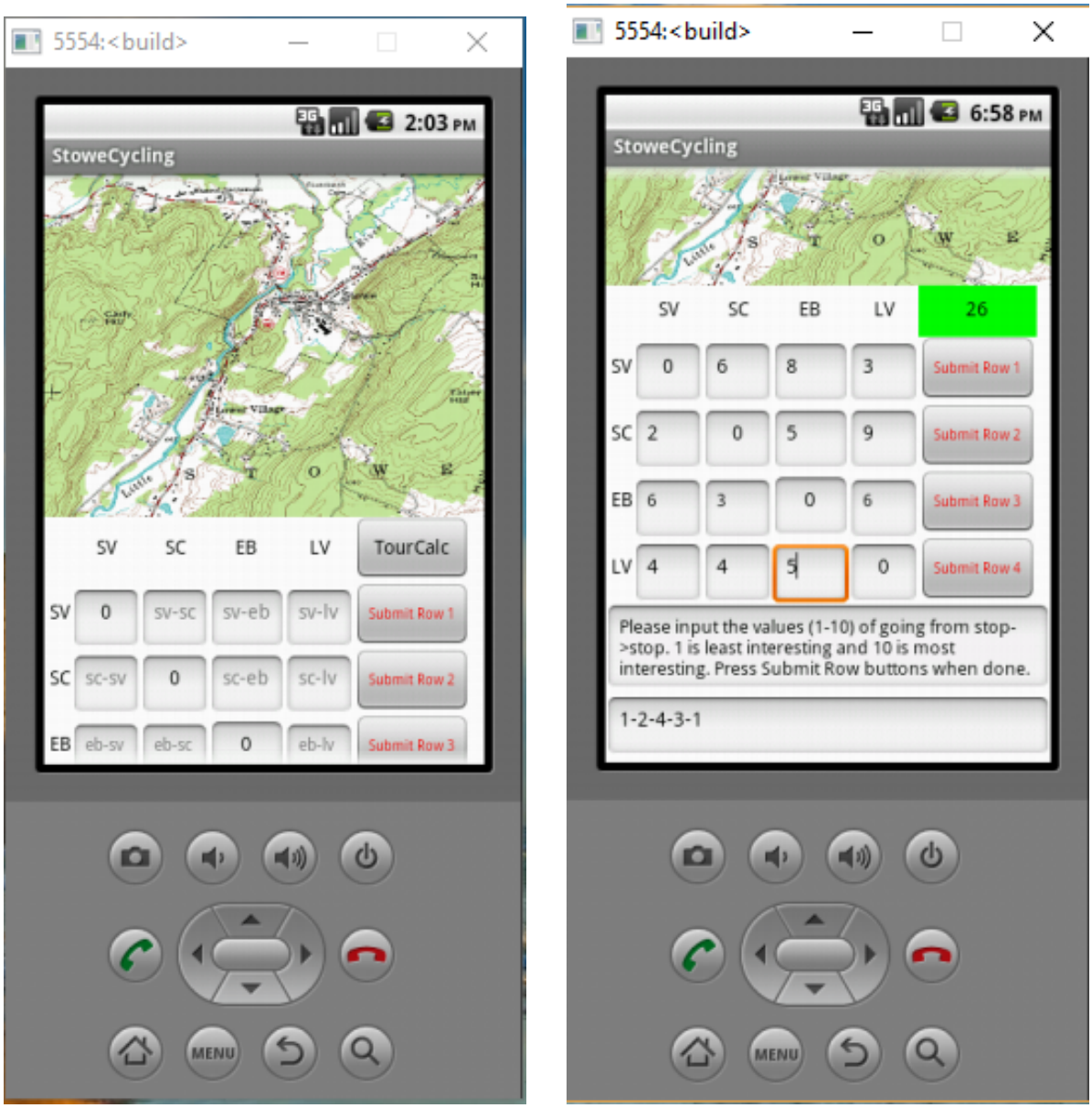


Figure 9. Stowe Cycle App Solution

We will see that this app relies on the strategy principles of sorting and simple comparisons to execute its logic. There can be many complex algorithmic and optimization procedures for this order picking problem as it involves location, time, the selection process, and the number of items to be picked.

2.2.1. Introduction The University of Massachusetts has a warehouse of food items that need to be delivered to the various dining commons, snack bars, and cafes on campus every morning before the dining operations get underway. Connor Tremarche a student in the Mie 379 class in 2012 worked at the warehouse and in order to make his life easier came up with the idea of programming the order pick app.

2.2.2. Problem Every morning before the day begins, employees of the warehouse have $10-15$ clipboards for which they must select items to be delivered on pallets to the dining services. Because this is a repetitious process occurring every morning, this type of phone app can be very important.

2.2.3. Mathematical Model Connor identified five key performance variables to be included in the app through a preference system on a scale of $1-4$ so he could deal with the different performance measure units (time, location, and number):

A Estimated delivery time (Time: 4 being the shortest)
B Location necessity (Rank: 4 being the highest)
C Number of locations (Exact Number)
D Estimated pick time (Time: 4 being the shortest)
E Estimated number of pallets (Exact number)

These performance criteria were to be matched with the alternative requesting facilities and the overall objective was to pick those items with the highest score. So in summary, we have the following preference ordering model where p_{ij} is the preference number of the performance variable and facility x_{ij} on the criterion:

$$\text{Maximize } Z = \sum_{i=1}^{5} \sum_{j}^{8} p_{ij} x_{ij} \tag{6}$$

$$s.t. \qquad x_{ij} \geq 0 \ \forall ij \tag{7}$$

2.2.4. Algorithm Figure 10 illustrates the blocks programming used in the app. Simple arithmetical operations are used.

2.2.5. Demonstration As can be seen in Figure 11 this is a well-designed app. The problem instance has the five criteria on the top and the eight different facilities with cells below. Once the input data are entered, the app sorts the orders into a pick order working from highest to lowest. Sorting is the key algorithmic process.

2.2.6. Evaluation The app design and blocks programming is very straightforward. It works very well. In the next section of the book, we examine what is called the Analytical Hierarchy Process (AHP) which is a technique for rank ordering a finite set of objects.

2.3. Analytical Hierarchy Process (AHP)

Many times we need to compare different alternatives and choose the best one, but we don't have a Linear Programming (LP) context in which to carry our the process. The context here can be considered as decision making problem under uncertainty rather than certainty in which LP occurs. So one procedure is to create a set of attributes/criteria to evaluate all the alternatives and use the criteria to select the best alternative. We do not *a priori* know the ranking of the criteria, so we must ask preference question to structure the ranking of the criteria. This is the essence of the app. The Analytical Hierarch Process (AHP) although sometimes controversial has become one effective way to rank criteria. It is discussed in Chapter 15 of Taha's book. We will provide a brief summary of the methodology.

2.3.1. Introduction T.L. Saaty has developed an approach to ranking objectives and attributes along with alternatives that utilizes the paired comparison approach in a very clever fashion. We will briefly summarize the methodology and illustrate with a small example, but there are further details (*i.e.* consistency calculations) but I will not delve into the theory of these, just present them.

Step 1.0: Construct a hierarchy of objectives/attributes and alternatives with as many levels as needed and a decision at the top. Figure 12 is an example of choosing a smart phone with two criteria/attributes and three alternatives smart phones.

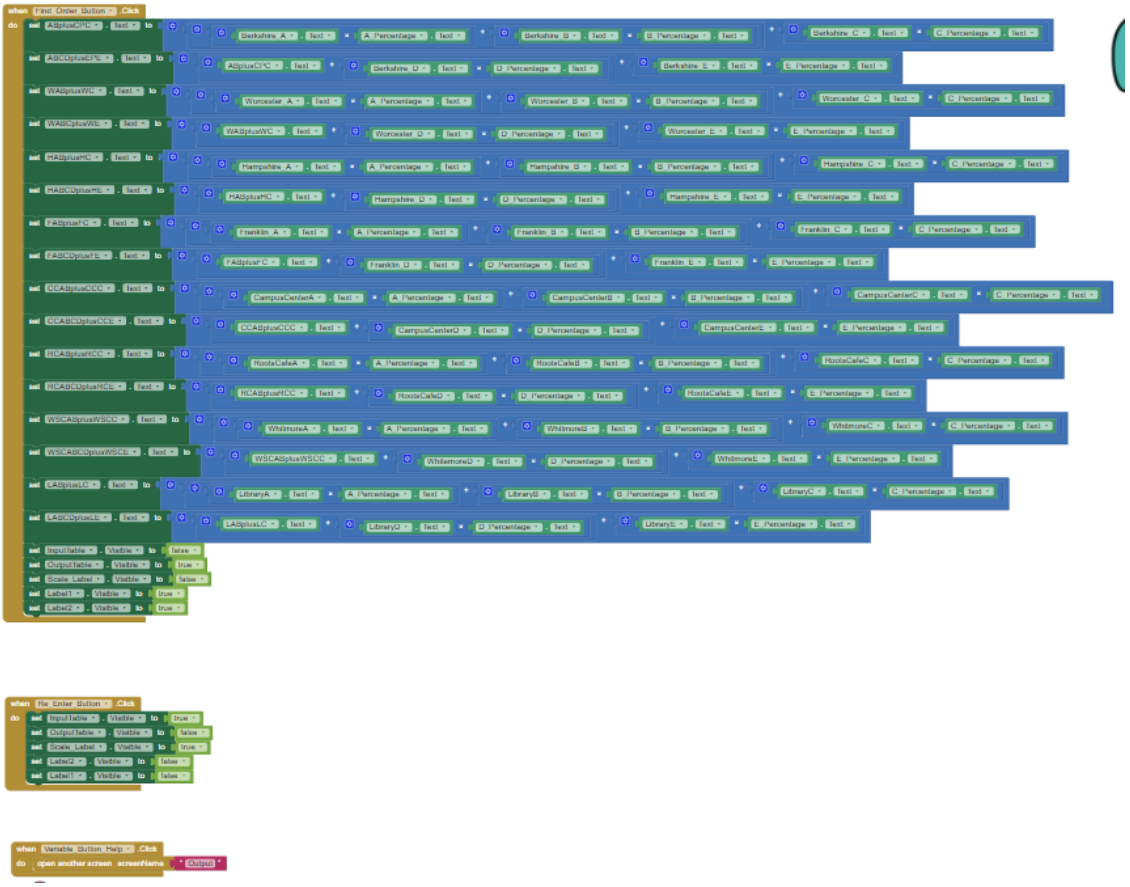


Figure 10. Warehouse Quick Pick Blocks

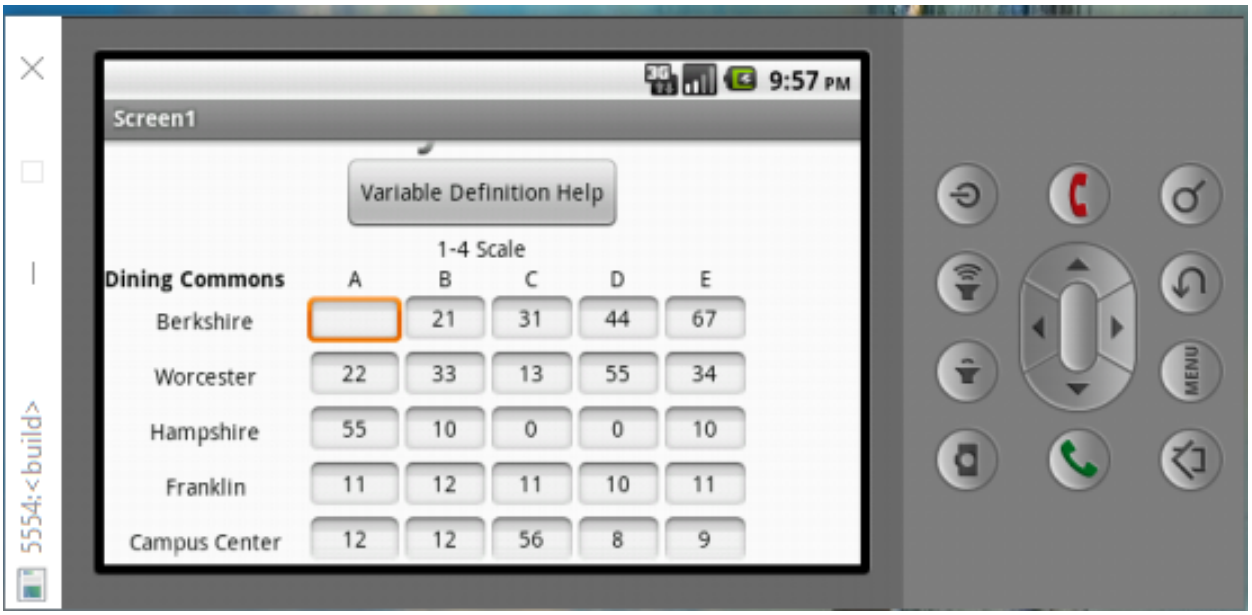


Figure 11. Warehouse Quick Pick App

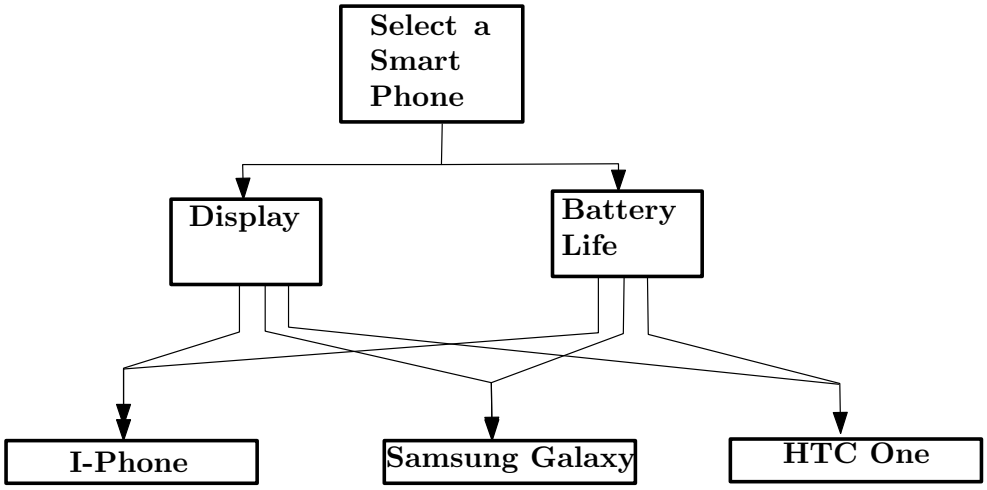


Figure 12. Example Hierarchy

Step 2.0: For each level of the hierarchy, first for the attributes/objectives, then the alternatives, construct a pairwise comparison matrix and evaluate each cell pair a_{ij} in the matrix according to the following continuous scale (interpolation is also valid):

Value a_{ij}	Comparison description
1	Attribute i and j are equally important.
3	Attribute i is weakly more important than j.
5	Attribute i is strongly more important than j.
7	Attribute i is very strongly more important than j.
9	Attribute i is absolutely more important than j.

$$
\begin{array}{c} \\ a_1 \\ a_2 \\ \vdots \\ a_r \\ \sum_i \end{array}
\begin{array}{c}
\begin{array}{cccccc} a_1 & a_2 & \dots & a_r & \sum_j \end{array} \\
\left[\begin{array}{ccccc}
a_{1,1} & a_{1,2} & \dots & a_{1,r} & \sum_j \frac{a_{1j}}{r} = w_1 \\
a_{2,1} & a_{2,2} & \dots & a_{2,r} & \sum_j \frac{a_{2j}}{r} = w_2 \\
\vdots & \vdots & \ddots & \vdots & \\
a_{r,1} & a_{r,2} & \dots & a_{r,r} & \sum_j \frac{a_{rj}}{r} = w_r \\
\sum_i a_{i1} & \sum_i a_{i2} & \dots & \sum_i a_{ir} & 1
\end{array}\right]
\end{array}
$$

Step 3.0: Find the column sums $\sum_i a_{ij}\ \forall\ j$ columns which is used to normalize the values of the criteria within the columns.

Step 4.0: Find the row sums $\sum_j a_{ij}/r\ \forall\ i$ rows which computes the weights for each alternative $a_i\ \forall i$ across the criteria.

Step 5.0: Do this for all pairs in the hierarchy and sum the final values across the criteria and alternatives. Everything should add up to 1.

Step 6.0: Consistency Check

6.1: Compute $\mathbf{Aw^t}$

6.2: Compute $\frac{1}{r}\sum_{i=1}^{r} \frac{i\text{th entry in } \mathbf{Aw^t}}{i\text{th entry in } \mathbf{w^t}} = r_{max}$

6.3: $CI = \frac{r_{max}-r}{r-1}$

6.4: $RI = \frac{1.98(r-2)}{r}$

6.5: $CR = \frac{CI}{RI}$, If $CR < .10$ ranking is consistent.

Example

Let's say we wish to buy a new smartphone such as an: A: Apple I-phone, B:Samsung Galaxy, C:HTC One. Let's further argue that two objectives/criteria are most important to us:

- Screen Display (D): pixel resolution, color quality, etc/
- Battery Life (B): # of hours, ease of re-charging, etc.

$$P = \begin{array}{c} \\ D \\ B \\ \sum_i \end{array} \begin{array}{c} \begin{array}{ccc} D & B & \sum_j \end{array} \\ \begin{bmatrix} 1 & 1/5 & \\ 5 & 1 & \\ 6 & 1.2 & \end{bmatrix} \end{array}$$

$$P = \begin{array}{c} \\ D \\ B \\ \sum_i \end{array} \begin{array}{c} \begin{array}{ccc} D & B & \sum_j \end{array} \\ \begin{bmatrix} .17 & .17 & .17 \\ .83 & .83 & .83 \\ 1.00 & 1.00 & 1.00 \end{bmatrix} \end{array}$$

So through our analysis, battery life is more important than the display screen. Just to illustrate another example and its ranking, now let's add a third attribute: Web Navigation (N): wi-fi capability, bluetooth, etc. The first matrix represents the scores for the three different attributes. The second matrix represents the calculations of the weights of the attributes by the AHP.

$$P_1 = \begin{array}{c} \\ D \\ N \\ B \\ \sum_i \end{array} \begin{array}{c} \begin{array}{ccccc} D & N & B & \sum_j \end{array} \\ \begin{bmatrix} 1 & 3 & 5 & \\ 1/3 & 1 & 3 & \\ 1/5 & 1/3 & 1 & \\ 23/15 & 13/3 & 9 & \end{bmatrix} \end{array}$$

$$P_2 = \begin{array}{c} \\ D \\ N \\ B \\ \sum_i \end{array} \begin{array}{c} \begin{array}{cccc} D & N & B & \sum_j \end{array} \\ \begin{bmatrix} 15/23 & 9/13 & 5/9 & 0.63341 \\ 5/23 & 3/13 & 1/3 & 0.26045 \\ 3/23 & 1/13 & 1/9 & 0.10614 \\ 1 & 1 & 1 & 1.00 \end{bmatrix} \end{array}$$

In this example demonstration, the ranking is reversed. Let' check the consistency of our new ranking. Carrying out the consistency check to give reassurance to our ranking of the criteria for buying the smart phone we carry out Step 6.0.

Step 6.0: Consistency Check
6.1: Compute $\mathbf{Aw^t}$

$$\begin{pmatrix} 1 & 3 & 5 \\ 1/3 & 1 & 3 \\ 1/5 & 1/3 & 1 \end{pmatrix} \begin{pmatrix} .63341 \\ .26045 \\ .10614 \end{pmatrix} = \begin{pmatrix} 1.97546 \\ 0.8000 \\ 0.3256 \end{pmatrix}$$

6.2: Compute $\frac{1}{r}\sum_{i=1}^{r} \frac{i\text{th entry in}\mathbf{Aw^t}}{i\text{thentry in }\mathbf{w^t}} = r_{max} = 3.101$

6.3: $CI = \frac{r_{max}-r}{r-1} = 0.05053$

6.4: $RI = \frac{1.98(r-2)}{r} = 0.66$

6.5: $CR = \frac{CI}{RI}$, If $CR < .10 \ = 0.0766 < 0.10$ranking is consistent
So, we are pretty comfortable with our current criteria ranking. For further details about the AHP please see Taha's Chapter 15. Now let's see how Amanda used the AHP to rank order the members of the marching band.

In marching band at the University of Massachusetts, there are a fixed number of marching spots available. This is done to create marching drill efficiently and on time. Occasionally, there are more candidates than drill spots so that the section leaders need to have a systematic procedure to choose the candidates.

2.3.2. Problem Amanda Skriloff a student in 2016 in the course and in marching band designed the app. She identified five performance criteria essential to the selection problem.

- [Playing Ability (PA):] Ability of a person to play scales, rhythm, and show music.
- [Marching Ability (MA):] Ability of a person to march in proper form and execution.
- [Class Conflicts (CC):] Missing rehearsals due to class conflicts is critical. Because the marching band meets every day, being at rehearsal is important.
- [Improvement Potential (IP):] Candidate's ability to improve and grows with practice.
- [Enthusiasm (E):] Overall enthusiasm of a person. How passionate and enthusiastic is the candidate.

2.3.3. Mathematical Model Pairwise comparisons are made for each of the attributes. There are $\frac{(n^2-n)}{2}$ or ten comparisons.

For Amanda's problem, she used a 5×5 paired comparison matrix is used to compare and rank order the criteria. The $\square$ is the one of the ten items to be answered in the app.

$$\begin{array}{c} \\ PA \\ MA \\ CC \\ IP \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} PA & MA & CC & IP & E \end{array} \\ \begin{bmatrix} -- & \square & \square & \square & \square \\ & -- & \square & \square & \square \\ & & -- & \square & \square \\ & & & -- & \square \\ & & & & -- \end{bmatrix} \end{array} \tag{8}$$

The ten total comparisons are made according to the following rules:

- If both attributes are equally important, a 1 is placed.
- If the attribute on the RIGHT is more important a number greater than 1 (3,6, or 9) is placed meaning that the item on the RIGHT is x times more important than the left item.
- If the attribute on the LEFT is more important a number less than one (1/3,1/6,1/9) is placed. meaning that the item on the left is y times more important than the right item.
- The matrix is then squared and there is a summation of each row and a grand summation of each row total. Each row total is then divided by the grand total. The highest number represents the most important attribute. The lowest number is the least important attribute

2.3.4. Algorithm Figure 13 illustrates all the blocks used for programming the app.

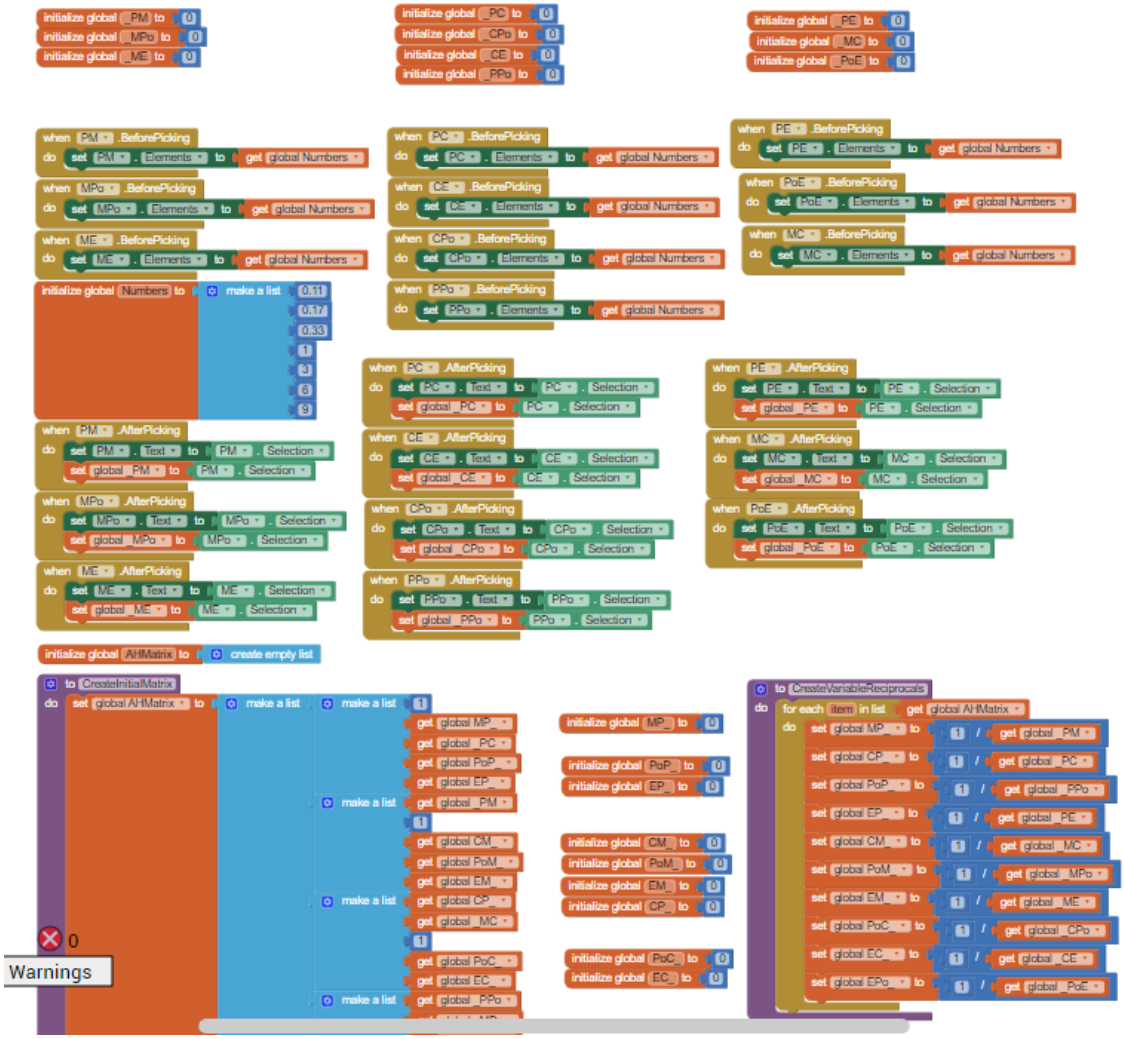


Figure 13. AHP Algorithm Blocks Programming

2.3.5. Demonstration As can be seen in Figure 14, the app works very well and is nicely designed.

2.3.6. Evaluation The app uses the methodology of the AHP to create a rank ordering of the criteria. It does not actually rank the candidates which is a downside, but the problem of coming up with the ranking of the criteria is a challenging first step to this process. It is a well designed app. Now, let's describe two applications where the theory is provided but the apps are still under development.

2.4. Shortest Path Problems

Shortest Path problems are classic combinatorial optimization problems and eminently suitable for a smart phone environment. They represent problems appropriate for a recursive Dynamic Programming (DP) approach since the "stages" of the DP approach are often considered the nodes of the $G(V, E)$ and the "states" are the remaining distance information functions at each node.

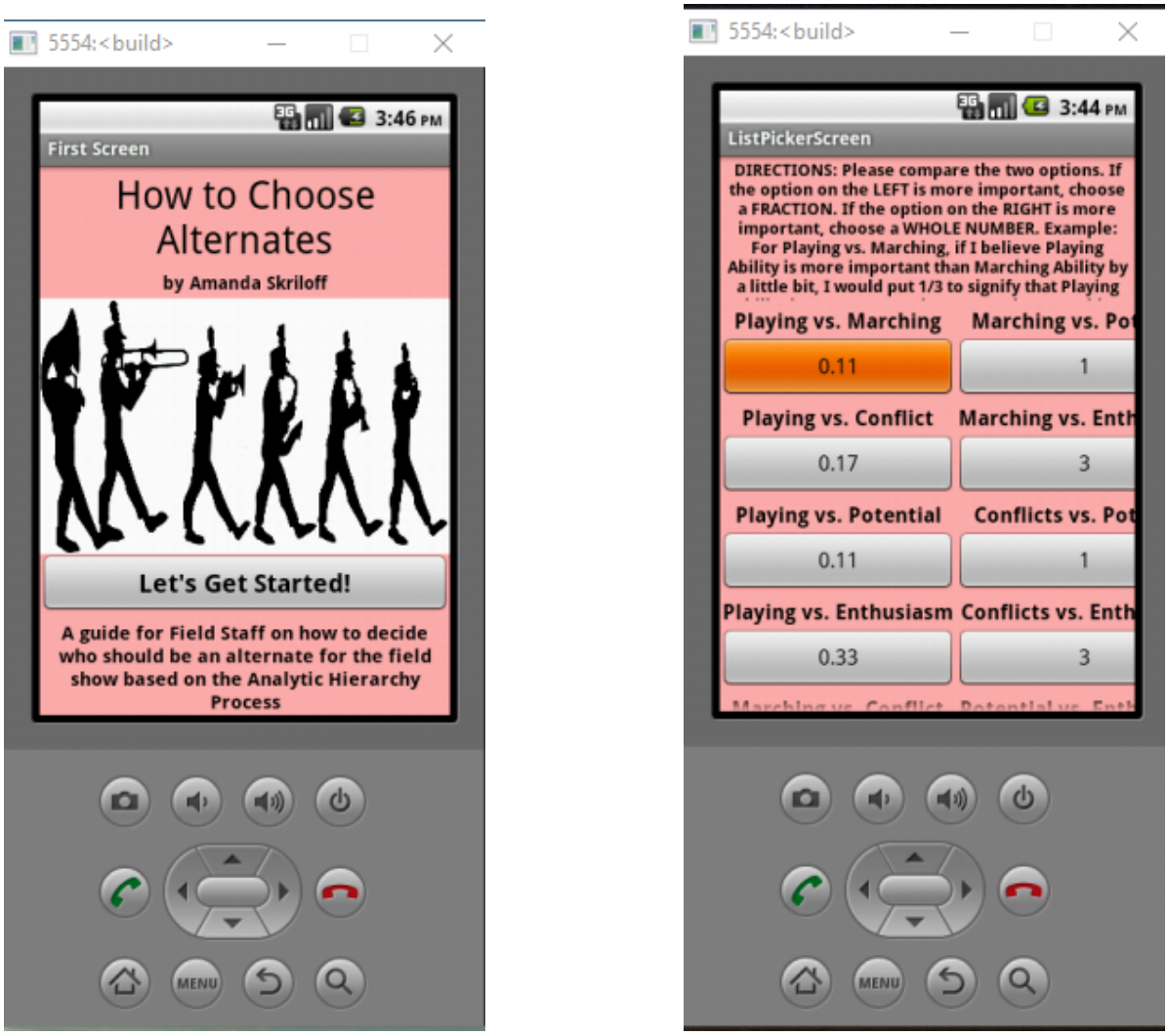


Figure 14. AHP AI2 Output Apps

2.4.1. Introduction

1. First of all, what we would like to find is that given a specified pair of nodes (i, j), determine the shortest path between i and j.

2. Secondly, we might want to find the shortest path tree between i and all other nodes in the graph, or even the set of shortest paths between all pairs on nodes in the graph.

3. Finally, we might want to find the shortest chains between all pairs of nodes in the graph.

• Figure 15 on the bottom left is that of a unit length dodecahedron mapped onto the plane. So the edge weights are all equal to 1.

• The figure next to it is the shortest path tree on the graph assuming we are starting at vertex #1 which is at the top of the graph and we wish to connect to all the nodes in the graph.

Example Applications include:

a) Manufacturing Process Planning
b) Equipment Replacement
c) Longest Paths (CPM/PERT Networks)
d) Most Reliable Routes
e) Bottleneck Arc Problems

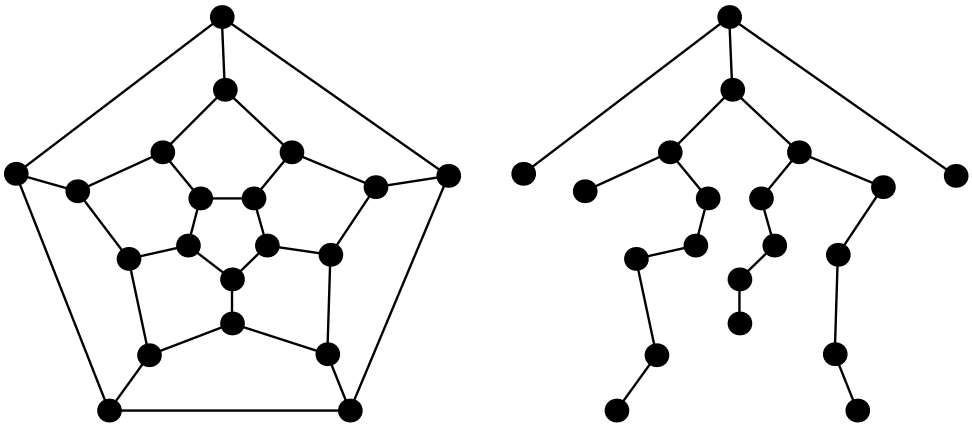


Figure 15. Dodecahedron Graph Shortest Path

2.4.2. Problem

Problem: Given an $G(N, A)$ with weight c_{ij} associated with each arc $a_i \in A$, find the shortest (minimal) chain between two specified nodes s and t in $G(N, A)$.

i) In general, we have the following assumptions which may be applicable to most any type of application as we shall illustrate:

ii) c_{ij} can be negative (certain application may allow for negative arcs *e.g.* profits)

iii) $c_{ij} = \infty$ if no arc exists between i and j.

iv) Negative circuits are not allowed.

v) $c_{ik} + c_{kj} \leq c_{ij}$ (a.k.a) the triangle inequality.

2.4.3. Mathematical Model We can formulate the shortest path problem as an LP. This model is based upon a min cost flow model of the shortest path problem. We are going to send one unit of flow from the starting node to the destination node which achieves the minimum cost flow.

Define

$$x_{ij} := \text{amount of flow in arc(i,j)} \tag{9}$$

$$= \begin{cases} 1 & \text{if arc } (i,j) \text{ is on the shortest route} \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

$$c_{ij} = \text{length of arc } (i,j) \tag{11}$$

Then, the objective function of the LP becomes the following linear objective subject to what are called the conservation of flow equations:

$$Minimize Z = \sum_{\text{all arcs} \in \text{G}} c_{ij} x_{ij} \tag{12}$$

$$\sum_{\text{all forward arcs} \in \text{G}} x_{ij} - \sum_{\text{all reverse arcs} \in \text{G}} x_{ij} = 0 \tag{13}$$

$$x_{ij} \geq 0 \tag{14}$$

2.4.4. Algorithm Recursive Dynamic Programming is the strategy that is used in this app program. It is a classic example of recursion.

Step 0.0: $[\delta_s] = 0; \delta_i = c_{si}; c_{si} = \infty$ (if no arc)

Set iteration counter $k = 1$

Step 1.0: $\delta_{last} = \min_{i \in \mathcal{L}} \delta_i$
$j = last :=$ the last node to get a permanent label
$\bar{\mathcal{L}} :=$ set of nodes with temporary labels.
Step 1.1: $[\delta_j] \leftarrow \delta_{last}$ and $\mathcal{L} = L \cup x_j; A = A \cup a_{ij}$
Step 1.2: If $[\delta_t]$ is found terminate.
Step 2.0: For each $x_i \in \bar{\mathcal{L}}$ replace δ_i with:

$$\delta_i = \min_{i \in \bar{\mathcal{L}}} \{\text{old } \delta_i; [\delta_{last}] + c_{ji}\}$$

Step 2.1: $k \longleftarrow k+1$
Step 2.2: Return to Step 1.0.

- $\sum_{k=1}^{N-1} 3(N-k) = 3[(N-1) + (N-2) + (N-3) + \ldots + (N-N-1)]$
- $\frac{3N(N-1)}{2} \Rightarrow O(N^2)$

2.4.5. Demonstration

2.4.6. Evaluation The next problem is concerned with Minimum Spanning Trees (MSTs). It is also still under development.

2.5. Minimum Spanning Tree Problems

Another classic combinatorial optimization problem concerns the construction of Minimum Spanning Trees (MSTs). These are problems appropriate for a Greedy Solution methodology. We shall examine Kruskal's algorithm. In Taha in Chapter 6, he presents Prim's algorithm. Both are greedy approaches and Kruskal's relies on sorting the edges of the graph to structure a solution.

2.5.1. Introduction We are given a graph and we wish to connect all the nodes together. An acyclic graph is one that consists of no cycles. A connected acyclic graph is called a tree. In a tree T, any two vertices (nodes) are connected by exactly one path. A subtree of a graph is a subgraph which is a tree. If the tree is a spanning subgraph, it is called a spanning tree.

2.5.2. Problem For an example, a telephone company wishes to rent a subset of existing cables each of which connects two cities. The rented cables should suffice to connect all cities and they should be as cheap as possible. A minimum spanning tree (MST) $\mathbf{T}$ which spans N nodes such that:

$$\sum_{(i,j) \in T} c_{ij} \text{ is a minimum}$$

There are many applications of MSTs.

o highway maintenance	**o cable tv networks**
o pipeline distribution	**o hvac systems in buildings**
o long distance telephone rates	**o airline routes**
o statistical clustering	**o molecular modelling**

2.5.3. Mathematical Model **[def:]** Given a Graph $\mathbf{G(N, A)}$ be an undirected connected graph. A subgraph $T' = (N, A')$ of G is a spanning tree of G if and only if T is a tree and connects all nodes in N. If each arc has a weight (cost) c_{ij}, then we seek to minimize the overall construction cost of the tree T.

2.5.4. Algorithm We shall follow Kruskal's algorithm for constructing the MST. which is probably the easiest to understand and probably the best one to execute if you are doing it by hand.

Kruskal's Algorithm:
Step 1.0: Sort the edges of G in increasing order by cost.

Step 2.0: Keep a subgraph $S \in G$, initially empty.
Step 3.0: For each edge $e \in G$ in sorted order
If the endpoints of e are disconnected in S
add e to S.
Step 4.0: return S.

Whenever we add an edge (u, v) it is always the smallest connecting the part of the subgraph with the rest of G, so it is always a part of the MST.

2.5.5. Demonstration

2.5.6. Evaluation The next Chapter describes the use of Linear Programming (LP) in app development. Most of the Mie 379 course will focus on Linear Programming methods because of their generality. There are a number of problem instances which use Linear Programming as their basis. We will review at least two problem applications as well as illustrate an app which uses the simplex algorithm to solve small problems.

3. Linear Programming

In general, LP has the following characteristics.

- An objective function $f(x_1, \ldots, x_n)$ of $(x_1, \ldots, x_n)$ is a **linear function** if and only if for some set of parameter constants $c_1, c_2, \ldots, c_n, f(x_1, \ldots, x_n) = c_1x_1 + c_2x_2 + \ldots + c_nx_n$. Some examples of linear and nonlinear objective functions are as follows:
  - $f(x_1, x_2) = 3x_1 + 2x_2$ is a linear function but $f(x_1, x_2) = 3x_1^2 + 2x_1x_2^2$ is not a linear function of x_1 and x_2.
- For any linear objective function $f(x_1, \ldots, x_n)$ and any right hand side number b, the inequalities which will constrain the solutions for our objective $f(x_1, \ldots, x_n) \leq b$ and $f(x_1, \ldots, x_n) \geq b$ are **linear inequalities.** Some examples are:
  - $2x_1 + 3x_2 \leq 4$ and $4x_1 - 3x2 \geq 1$ are linear inequalities but $x_1^3x_2 - 2x^2 \geq 3$ is not.
- A linear programming problem (LP) is an optimization problem for which we do the following:

1. We attempt to maximize (or minimize) a linear function of the decision variable. This function is called the objective function.
2. The values of the decision variables must satisfy a set of constraints. Each constraint is either an inequality or an equality.
3. A sign restriction is associated with each decision variable. For any x_i, the sign restriction specifies that x_i must be either nonnegative $(x_i \geq 0)$ or unrestricted in sign (urs).

So if we assume these linear relationships, then we can formulate the following LP model. We need some basic definitions of the elements of the LP. These include:

- *Decision Variables:* Usually denoted as vector of variables with subscripts although they need not be subscripted $\mathbf{x} = (x_1, \ldots, x_n)$
- *Objective Function:* A performance variable that is either minimized or maximized and usually denoted as $f(\mathbf{x})$
- *Constraints:* A qualitative objective that is usually a bounding mechanism on the decision variables, usually denoted as $g(\mathbf{x}) \leq 0; h(\mathbf{x}) = 0$

Once we have these component elements, then we can formulate algebraically a general mathematical programming problem model as:

$$\text{Max or Min } f(\mathbf{x})$$

subject to:

$$\text{inequalities } g_i(\mathbf{x}) \leq 0 \tag{15}$$
$$\text{equalities } h_i(\mathbf{x}) = 0 \tag{16}$$
$$\text{nonnegativity } \mathbf{x} \geq 0 \tag{17}$$

Some of the key assumptions one must make in setting up LPs are the following:

- **Proportionality:** the objective functions and constraints are linear in the given parameters.
- **Additivity (Separability)**: the decision variables are separable and independent.
- **Divisibility** the decision variables are continuous.
- **Certainty** the values of the objective function and constraints and parameters are know with certainty.

Given these assumptions and the general linear programming structure, let's examine some LP apps. Normally, these liner programs are solved with what is called the simplex algorithm. The apps described in the following pages will not always use the simplex algorithm for their solution but rely on the linear properties for their solution.

3.1. Equalization of Runout Times (ERT) Problem

A closely related problem to the Stowe Cycle TSP routing problem which we reviewed earlier is concerned with an inventory resource allocation problem where the allocation of the resource to the demand points $V = \{v_1, v_2, \ldots, v_n\}$ is such that it maximizes the time at which the group of demand points will next be scheduled. This is actually classified as a capacitated inventory problem on a graph which turns out to be a linear programming problem.

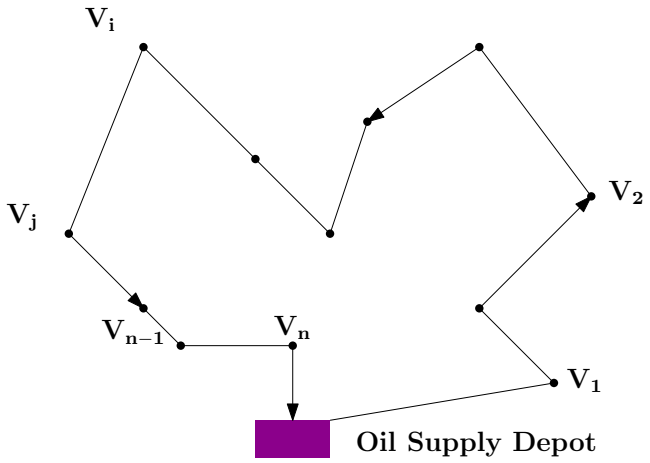


Figure 16. ERT Oil Supply Problem

3.1.1. Introduction For example, suppose we have a heating oil delivery problem and we wish to allocate the amount of oil to each of $n-$households with demand point x_i so that the allocation of oil will ensure the maximum time to re-supply, *i.e.* all households will run out of oil at the same time. If we don't have the households running out of oil at the same time, then the truck must be dispatched to the households too many times, whereas if they run out at the same time, then we can dispatch the truck once. What will be the allocation of oil to each household so that you maximize the time between deliveries.

3.1.2. Problem Fundamentally, this is a Linear Programming (LP) problem and is one of the basic set of topics in the Taha textbook which we will study, but actually we can solve it more readily without the simplex method. Below are some of the variables and parameters we need to build a mathematical model of the problem.

3.1.3. Mathematical Model The notation needed for the ERTGO problem is described below.

Variables	Description
$D_i :=$	Demand at household i
$E_i :=$	Initial inventory supply at the household
$K :=$	Capacity of Delivery Vehicle
$n :=$	n-households
$t :=$	Time to runout of the inventory item and for each household $t_i = x_i/D_i)$
$x_i :=$	Decision variable for the allocation at a household

Our Linear Programming (LP) formulation appears in the following statements where we maximize the runout time subject to the demand and current inventory levels and the capacitated supply constraints of the perishable item. In the formulation below, the initial inventory level is 0.

O Objective function:

$$\text{Maximize } t$$

C Constraints:

$$\sum_i^n x_i = K \tag{18}$$

$$\frac{(x_i + E_i)}{D_j} - t \geq 0 \tag{19}$$

$$x_i \geq 0 \; \forall i \tag{20}$$

If in addition, we allow for an initial inventory, then we have the following extension:

- $w_i :=$ time to run out with the current inventory ($w_i = E_i/D_i$)
- $u_i :=$ time to run out with the new allocation ($u_i = x_i/D_i$);

Then a new problem can be formulated as an LP:

O Objective function:

$$\text{Maximize } t$$

C Constraints:

$$\sum_i^n u_i D_i = K \tag{21}$$

$$w_i + u_i - t \geq 0 \tag{22}$$

$$u_i, w_i, t \geq 0 \; \forall j \tag{23}$$

In general, LP must be used, but if the initial inventory is relatively small, the ERT rule works:

$$w_i + u_i = t \rightarrow u_i = t_i - w_i, \text{and } \sum u_i D_i = \sum (t - w_i) D_i = t \sum D_i - \sum w_i D_i = K$$

and

$$t = (H + \sum w_i D_i)/(\sum D_i) = (K + \sum E_i)/(\sum D_i) = t$$

and

$$u_i = t - w_i \rightarrow (K + \sum E_i)/(\sum D_i) - E_i/D_i$$

then the optimal allocation for a household is:

$$x_i = D_i t - E_i$$

and this allocation process will work as long as $x_i \geq 0$

The app can be directly programmed as described in the next section of the book, since the solution methodology is relatively straightforward and depends upon the data about the demands and inventory and the formula above.

3.1.4. Algorithm We do not have to use the simplex algorithm of Linear Programming per se to solve the problem. The structure of the problem while a Linear Program admits to a straight-forward solution. Figure 17 illustrates a sample of the blocks programming of the ERT app. The blocks detail the summing of the demands. Because of the size of the designer screen, seven households can be the maximum used in the problem. A different screen design could obviously handle more demand points. The remaining blocks are available with the app on the website.

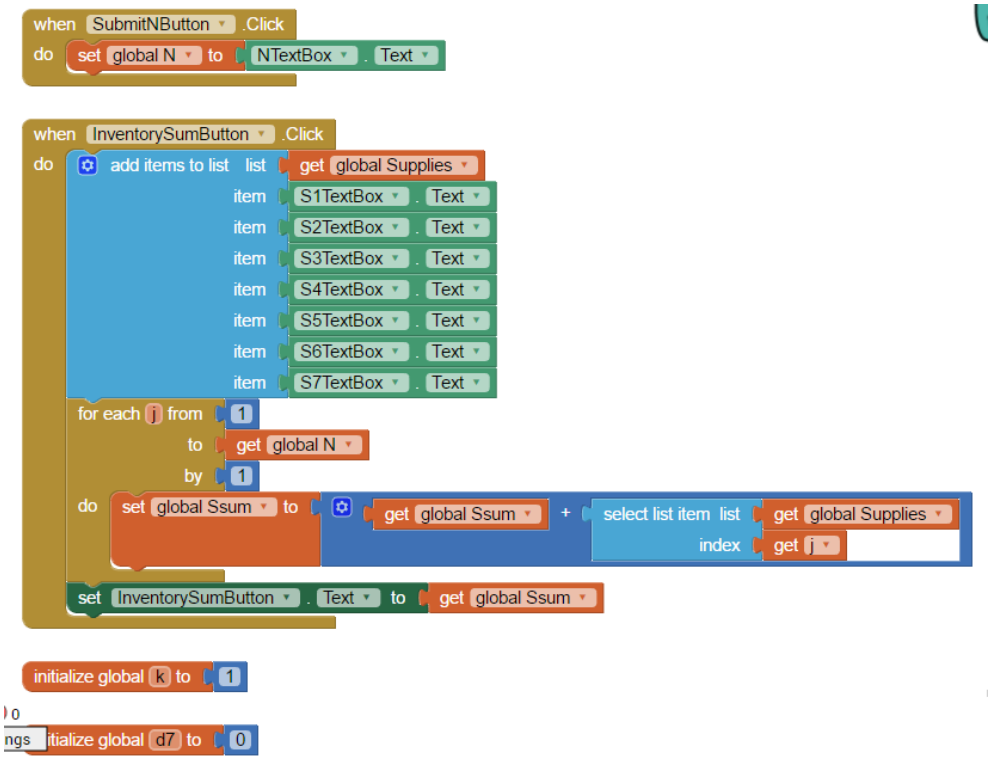


Figure 17. ERT Blocks

3.1.5. Demonstration Figure 18 illustrates the final implementation of the ERTGO app along with a demonstration for the following data and parameters. For example, let's say that we have $N = 4$ and the demand vector for the households is: $D = \{120, 180, 900, 50\}$; the current inventory supply is $E = \{30, 120, 300, 20\}$; and we have $K = 1000$ gallons capacity for the truck, Running the app, we find that $t = 1.1760$ time periods and the allocation to the households is $X = \{111.12, 91.68, 758.4, 38.8\}$. It is very nice that the algorithm takes the capacity of the truck and fills the household demands very compactly.

3.1.6. Evaluation The app works very well. For another application where LP is not directly used, we discuss a location problem.

3.2. Pinball Weber

This is a location problem with a rectilinear (taxicab) metric which can be formulated as a Linear Program. The solution will not explicitly use LP per se, but will use the separability properties and nature of the problem to generate a methodology for solution.

3.2.1. Introduction The Steiner/Weber problem is a famous location problem that goes back to the 18^{th} century. In fact, I worked on this problem for my Ph.D. dissertation. Most location problems are nonlinear in nature since they require a Euclidean distance function. We wish to locate a new facility in relation to some existing facilities where we wish to minimize the total distance travelled between the new facility and the existing facilities.

3.2.2. Problem Let's say that we want to locate a new machine within a factory where we already have four existing machines located at the Cartesian points $(8, 5), (4, 2), (11, 8)$ and $(13, 2)$. In the manufacturing facility, there are weights of importance representing $\mathbf{w} = \{9, 6, 4, 12\}$ the expected number of trips per week for a material handling system between the new machine and the current machines. Minimizing the total weighted traffic flow is our objective.

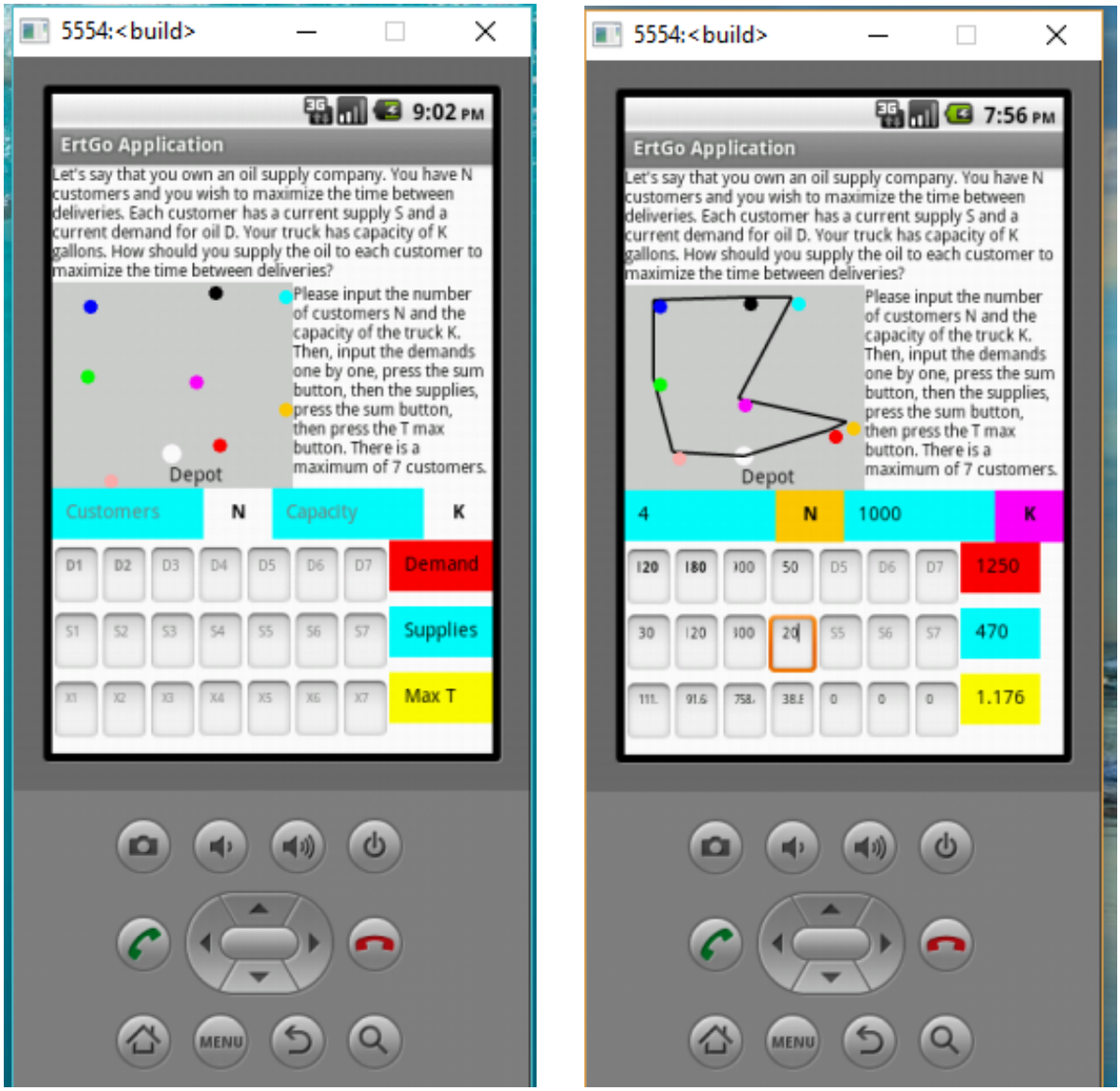


Figure 18. ERTGO App Success

3.2.3. Mathematical Model We wish to find the location of the new machine so as to minimize the total distance traveled between the new machine and the existing machines. Let's first analyze the problem where we assume Euclidean distance is used to travel between the new machine $i(x, y)$ and the $j-$existing machines.

$$\text{Minimize Z } = \sum_j w_j dist(i, j)$$

So we will also have an unconstrained optimization problem but instead of one variable we will have two variables (x, y). In a subsequent model, we will use rectilinear distance.

The objective function is:

$$Z = 9\sqrt{x^2 - 16\,x + 89 + y^2 - 10\,y} + 6\sqrt{x^2 - 8\,x + 20 + y^2 - 4\,y} + 4\sqrt{x^2 - 22\,x + 185 + y^2 - 16\,y} + 12\sqrt{x^2 - 26\,x + 173 + y^2 - 4\,y}$$

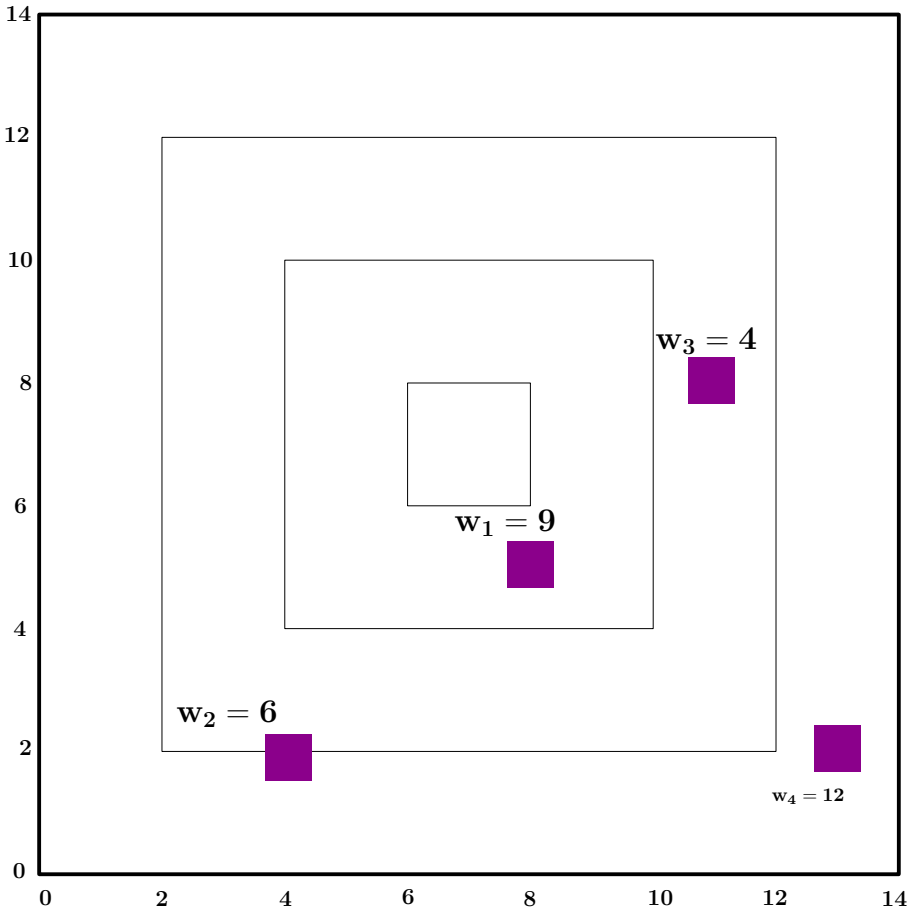


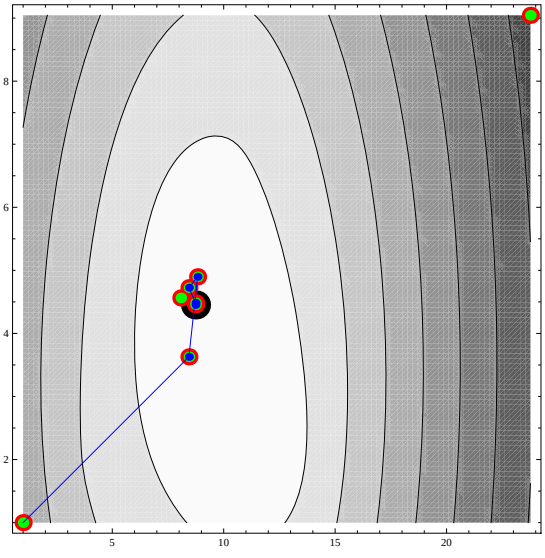
Figure 19. Layout of Factory

The derivatives are very complicated and the nonlinear equations cannot be separated in the decision variables x, y so we have the following:

$$-\frac{9\,(2\,x-16)\,(2\,y-10)}{4\,(x^2-16\,x+89+y^2-10\,y)^{3/2}}-\frac{3\,(2\,x-8)\,(2\,y-4)}{2\,(x^2-8\,x+20+y^2-4\,y)^{3/2}}$$
$$-\frac{(2\,x-22)\,(2\,y-16)}{(x^2-22\,x+185+y^2-16\,y)^{3/2}}-\frac{3\,(2\,x-26)\,(2\,y-4)}{(x^2-26\,x+173+y^2-4\,y)^{3/2}}$$

The actual solution is (via an unconstrained optimization algorithm)

$$[Z=116.117,[X=8.737,Y=4.451]]$$



Here, Mathematica was used to find the solution.

Optimization‘UnconstrainedProblems‘

$$f=9\sqrt{x^2-16x+89+y^2-10y}+$$
$$6\sqrt{x^2-8x+20+y^2-4y}+$$

$4\sqrt{x^2-22x+185+y^2-16y}+$
$12\sqrt{x^2-26x+173+y^2-4y}$

FindMinimumPlot[$f,\{\{x,1\},\{y,1\}\}$, Method $\rightarrow$ Newton] $\{\{116.117,\{x \rightarrow 8.73765, y \rightarrow 4.4518\}\},\{"Steps" \rightarrow 7, "Function" \rightarrow 10, "Gradient" \rightarrow 10\}$

Actually, if we change our distance function from Euclidean to rectilinear, our problem greatly simplifies.

$$\text{Minimize Z } = \sum_j w_j(|x-a_{j1})| + |y-a_{j2}|)$$

This problem is separable in x, y the decision variables:

$$Z_x = \sum_j w_j(|x-a_{j1}|)$$

$$Z_y = \sum_j w_j(|y-a_{j2}|)$$

We actually can show that these two separate functions are convex and the optimal (x, y) location is the median sum of the weights of the existing sorted facilities on the x and y coordinates. Let's take our example problem and examine this convexity issue:

Original Coordinates				sorted x_1 dimen.		sorted x_2 dimen.	
j	a_{j1}	a_{j2}	w_j	a_{j1}	w_j	a_{j2}	w_j
1	8	5	9	4	6	2	6
2	4	2	6	8	9	2	12
3	11	8	4	11	4	5	9
4	13	2	12	13	12	8	4

$$W_1(x_1) = 6|x_1-4| + 9|x_1-8| + 4|x_1-11| + 12|x_1-13|$$

$$W_1(0) = 296, W_1(4) = 145, W_1(8) = 96, W_1(11) = 93, W_1(13) = 107$$

$$W_2(x_2) = 6|x_2-2| + 12|x_2-2| + 9|x_2-5| + 4|x_2-8|$$

$$W_2(0) = 113, W_2(2) = 51, W_2(5) = 66, W_2(8) = 135$$

3.2.4. Algorithm The algorithm solves the problem for the X-coordinate then the Y-coordinate because the problem is separable in the decision variables. It is very straightforward.

- [Step 1.0:] First find the sum of the weights $\sum_j^n w_j$
- [Step 2.0:] x^* is the first a_j (sorted x-order) at which the cumulative weights reaches or exceeds the median value $\sum_j^n w_j/2$
- [Step 3.0:] y^* is the first a_j (sorted y-order) at which the cumulative weights reaches or exceeds the median value $\sum_j^n w_j/2$

In our example problem, the sum of the weights is $W = 31$ (*i.e.* median = 15.5). We sort on the x-axis first, identify the

- $x = \{4, 8, 11, 13\}$, and $w_j = \{6, 9, 4, 12\}$ then $x^* = 11$
- while $y = \{2, 2, 5, 8\}$ and $w_j = \{12, 6, 9, 4\}$ then $y^* = 2$,
- the optimal location is $(\mathbf{x}, \mathbf{y}) = (\mathbf{11}, \mathbf{2})$ with $Z = 144$.

3.2.5. Demonstration Below in Figure 20 is the app and its block programming with AI2. The app is actually programmed to solve randomly generated problems and to show the graphic capabilities of the AI2 language and phone properties.

Figure 21 is the final realization of the app. The red ball represents the optimal location for the given problem, while the black dots represent the existing facilities. The green dot moves around the screen to show changes to the screen after the input arrangement of black dots is re-set. Once the green dot hits a boundary, the new optimal location appears as the red dot.

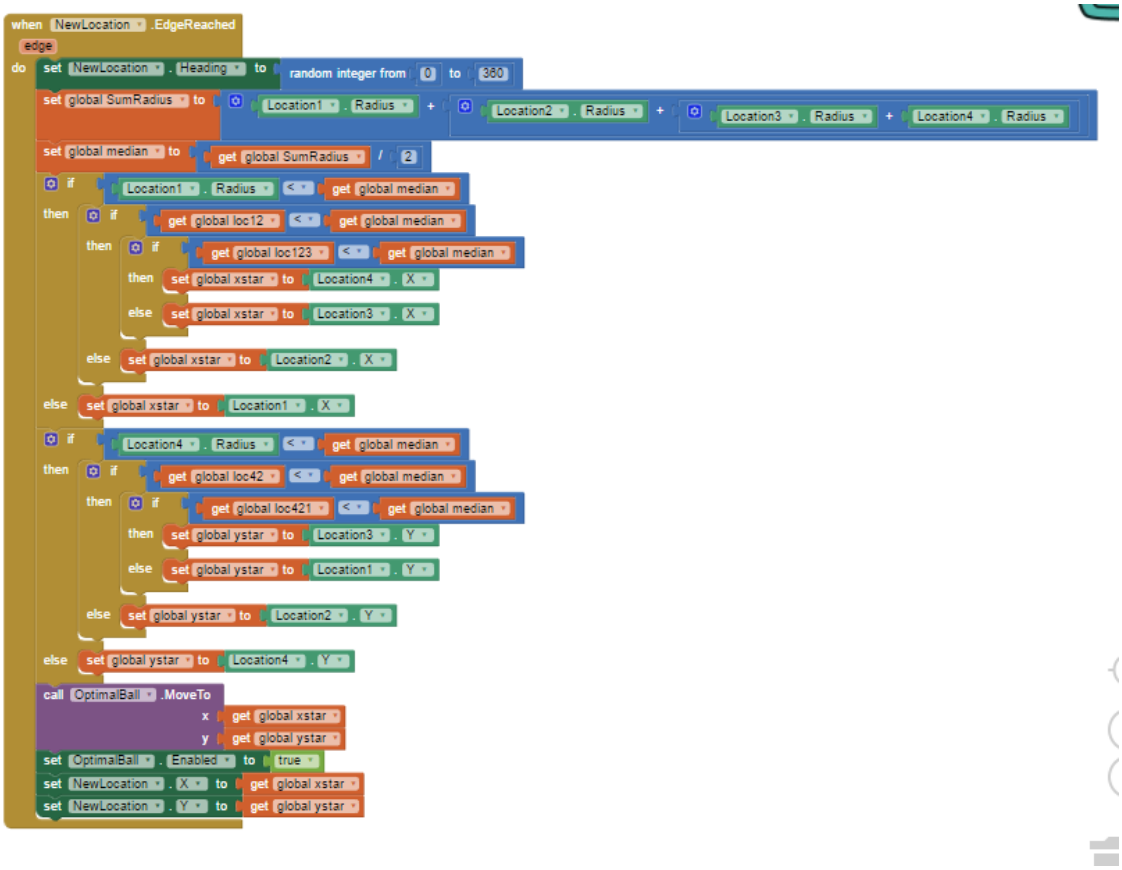


Figure 20. Sample Blocks and Pinball App

3.2.6. Evaluation Some of the features of the app are described below:

- Iteratively generates a new 4-facility location problem with random Cartesian coordinates and weights. It also recomputes the optimal location as the red dot on the screen. One of the nice features of App Inventor is the dynamics possible with the screen.
- ReStart button once pushed ⇒ randomly generates a new problem.
- Once green ball hits boundary, a new solution is found.
- The App works very fast.

The app could be programmed to handle a larger number of given facilities, and the choice of four is simply illustrative. The next app is related to the location app just presented but concerns the detailed allocation of plants to a garden plot.

3.3. Garden Planting

Here we will demonstrate the use of Linear Programming (LP) to aid a gardener in what type of plants to place in their garden. This is a classic form of resource allocation planing with LP. In one sense this is an Integer programming (IP) problem because of the integer nature of the planted items, but we will approximate this with continuous variables as an LP and allow for partial plants. Rounding up is also a viable approach for this type of resource allocation problem. We want to demonstrate the use of basic feasible solutions to linear programs.

3.3.1. Introduction Small scale gardening has become more popular in recent years as hobbyists and amateurs try to save money by planting herbs and vegetables in backyards and window boxes. The idea of the app is to assist gardeners in what to plant.

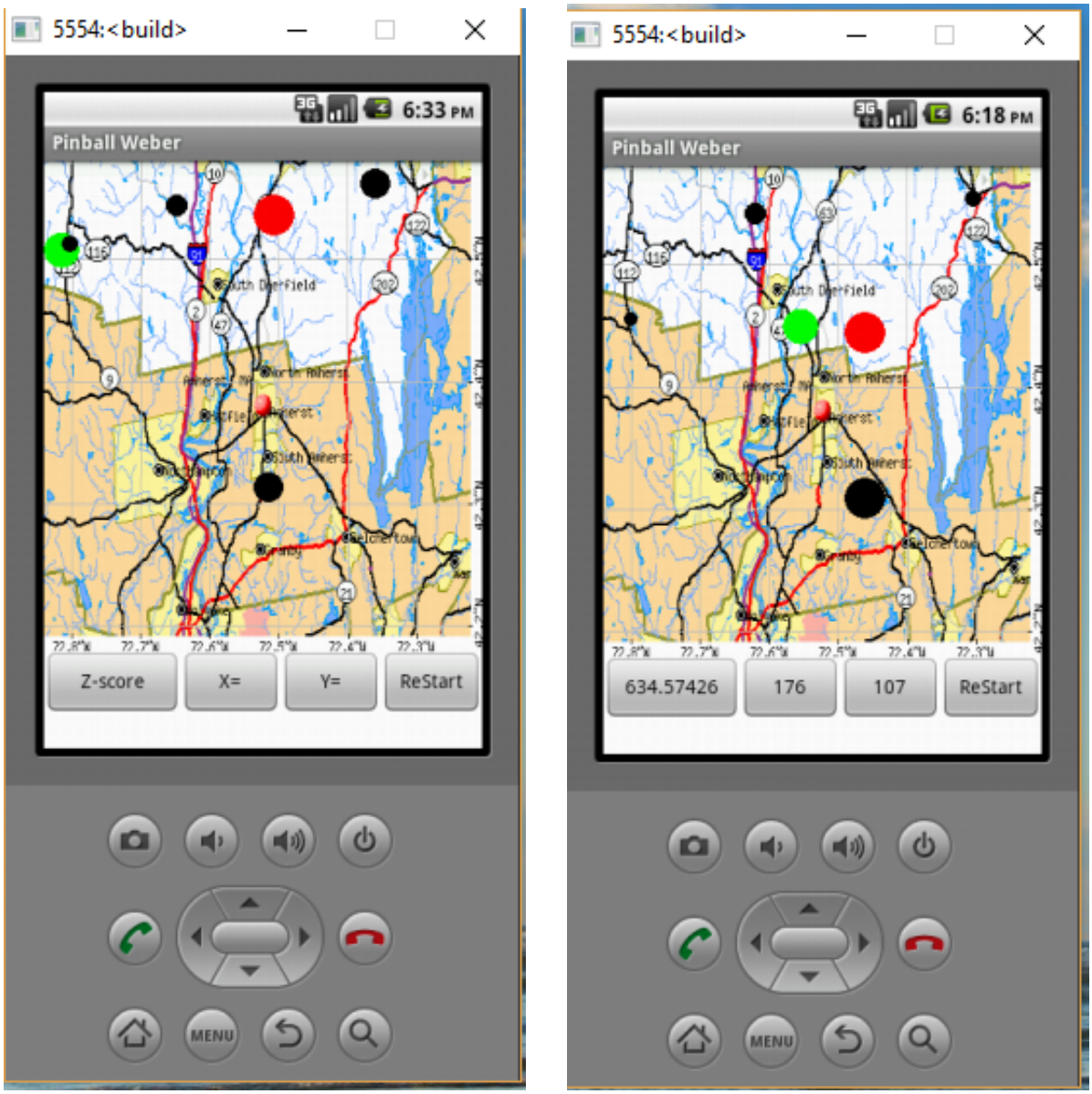


Figure 21. Pinball App Input and Solution Screens

3.3.2. Problem We assume that the gardener can either plant partial plants or start from seeds. Seeds are bought individually. We are assuming a single 4 planing box that has 16square feet of total area. We have information of the area requirements for each plant type. Thomas Johnson designed the app in 2014. There are eleven different decision variables or plant types:

- $x_{BR} :=$ Broccoli plants
- $x_{LL} :=$ Leaf Lettuce plants
- $x_{Ce} :=$ Cabbage plants
- $x_{SC} :=$ Swiss Chard plants
- $x_{Pe} :=$ Pepper plants
- $x_{BB} :=$ Bush Bean plants
- $x_{Ct} :=$ Carrot plants
- $x_{Sp} :=$ Spinach plants
- $x_{Ra} :=$ Radish plants
- $x_{Be} :=$ Beet plants
- $x_{On} :=$ Onion plants

3.3.3. Mathematical Model The LP is limited because the AI2 programming blocks were used to identify fifteen different basic feasible solutions. Of course the full app could be programmed to take advantage of the NEOS server to solve larger problems and that remains a good extension problem for future examination.

$$\text{Maximize } Z = \sum_j c_j x_j \tag{24}$$

$$s.t. \sum_j a_j x_j \leq A \qquad \text{area requirements} \tag{25}$$

$$\sum_j c_j x_j \leq B \qquad \text{budget constraint} \tag{26}$$

$$x_j \geq 0 \; \forall j \qquad \text{plants} \tag{27}$$

where
c_j:= cost per item planted
x_j:= continuous variable for item planted
a_j:= area requirements of item to be planted
A:= area of the planting box;
B:= the planting budget.

3.3.4. Algorithm The app examines all the basic feasible solutions so it is an enumeration strategy. Figure 22 illustrates some of the AI2 programming blocks.

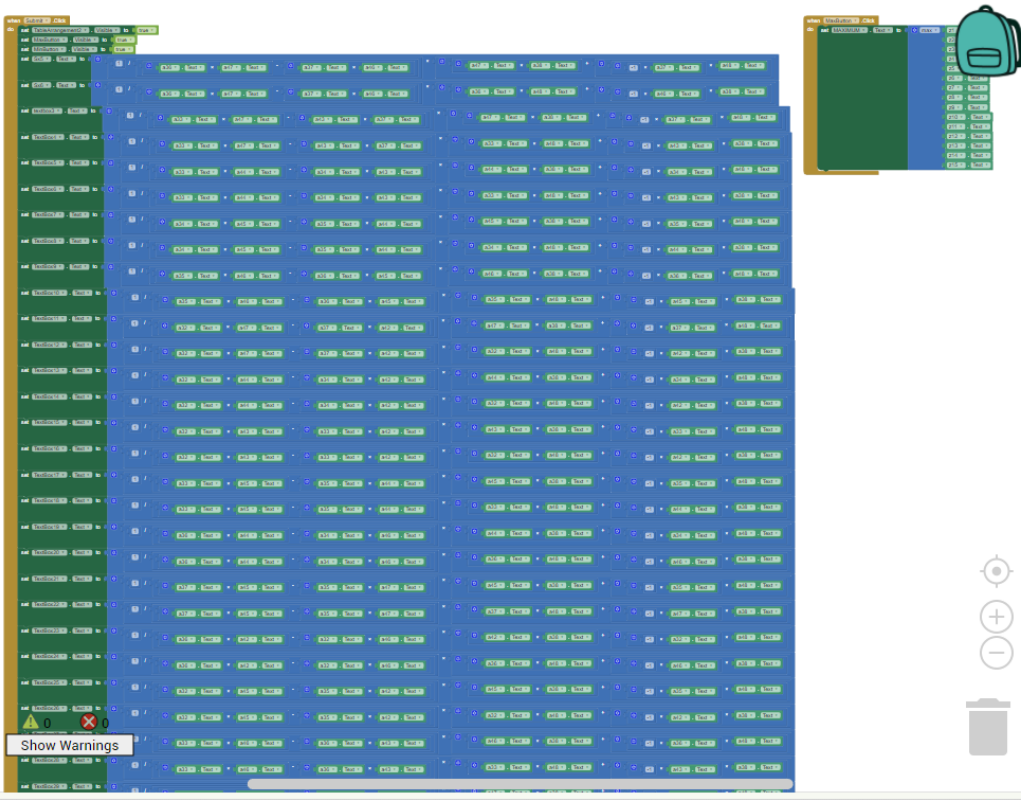


Figure 22. Garden Algorithm Blocks Programming

3.3.5. Demonstration As an example demonstration, we have the following data:

$$\text{Maximize } Z = 0.60x_{Br} + 4.05x_{LL} + 2.24x_{Ce} + 1.535x_{sc} + 3.6x_{Pe} + 0.279x_{BB} + \quad (28)$$
$$0.223x_{Ct} + 0.20x_{Sp} + 0.389x_{Ra} + 0.099x_{Be} + 0.259x_{On} \quad (29)$$
$$s.t.\ 1x_{Br} + 0.25x_{LL} + 1x_{Ce} + 0.25x_{Sc} + 1x_{Pe} + 0.111x_{BB} + 0.063x_{Ct} \quad (30)$$
$$+ 0.111x_{Sp} + 0.063x_{Ra} + 0.111x_{Be} + 0.063x_{On} \leq 16 \quad (31)$$
$$0.050x_{Br} + 0.009x_{LL} + 0.10x_{Ce} + 0.031x_{Sc} + 0.007x_{Pe} + 0.091x_{BB} \quad (32)$$
$$+ 0.008x_{Ct} + 0.020x_{Sp} + 0.025x_{Ra} + 0.050x_{Be} + 0.010x_{On} \leq 100 \quad (33)$$
$$x_j \geq 0\ \forall j \qquad \text{plants} \quad (34)$$

In solving this example demo, we find that the Lettuce Leaf $x_{LL} = 64$ plants is the best plant alternative and it fills the entire square footage available of the planting block. Figure 23 illustrates the app and its solution for the sample data. It works very well and is very fast. Because there are only two constraint inequalities here, it is not unheard of to have only a few decision variables in the optimal basis.

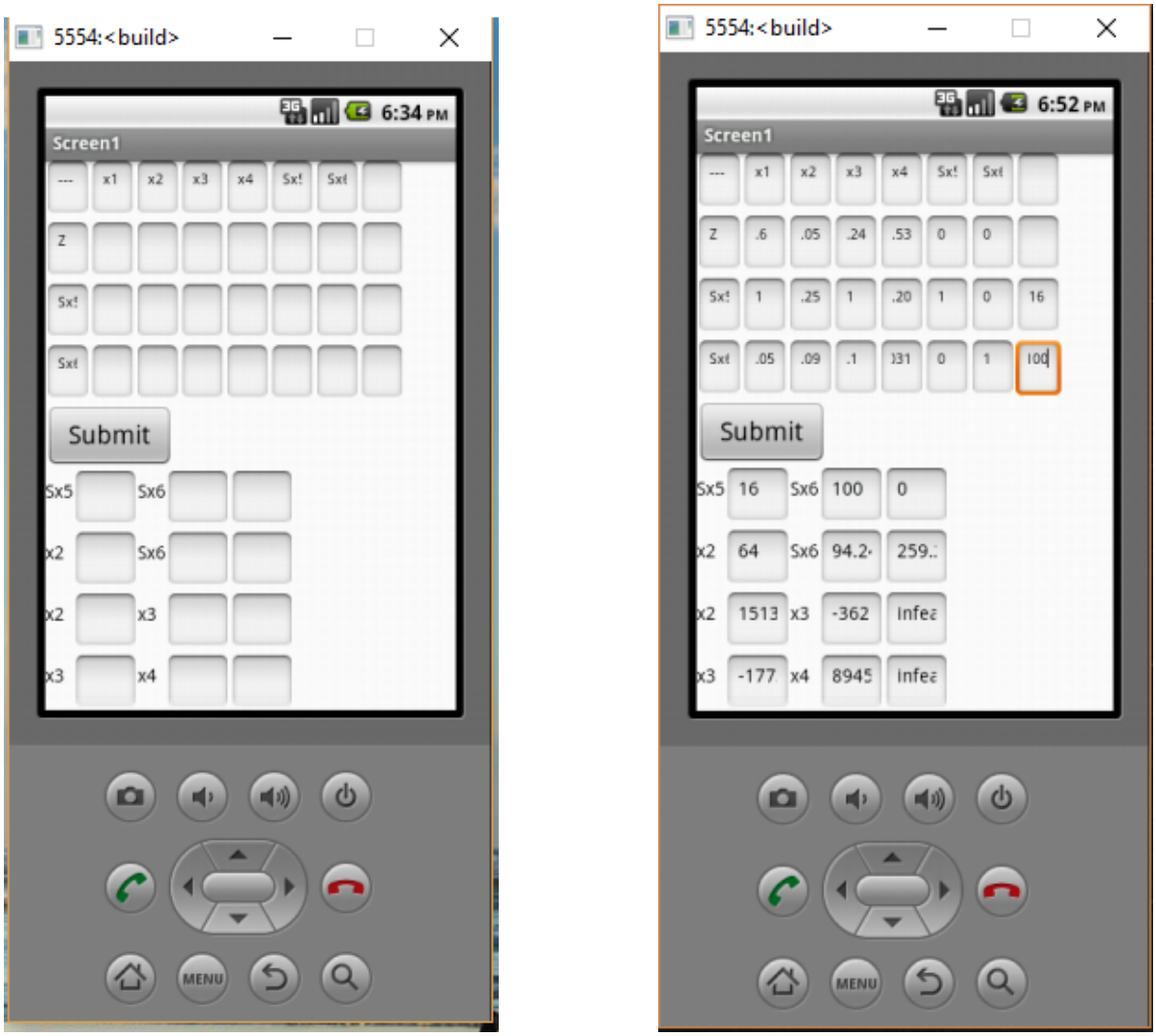


Figure 23. Garden App Programming and Sample Solution

3.3.6. Evaluation While this is a limited app because it does not require many decision variables or constraints, it demonstrates a nice application and usefulness of AI2 for enumerating LP solutions. Use of the NEOS server for larger LPs will be demonstrated later for a larger farm planning problem. Now let's examine an app which is designed to demonstrate the simplex algorithm itself.

3.4. Simplex and Dual Simplex Problems

We use a special implementation of the AI2 programming language to demonstrate the simplex and dual simplex algorithms. We will only examine small problems with a few decision variables, but their implementation in AI2 is worthy of examination. The student who programmed AI2 is Allan Tang who was a student in Computer Science at UMass in 2011. He programmed this in the first class that I taught with App Inventor so it was very daring at the time and is quite an accomplishment with initially App Inventor Classic. It has been converted to AI2.

3.4.1. Introduction The motivation is to demonstrate that the AI2 program is capable of solving some simple problems but also showing the details of the calculations of the simplex algorithm in action by showing the simplex tableaus.

3.4.2. Problem The problem for this app was to demonstrate the simplex algorithm and the dual simplex in an app setting. Allan wanted to show that it was possible to solve small LPs with App Inventor and demonstrate the simplex tableaus which are challenging for students to understand.

3.4.3. Mathematical Model One possible formulation of an LP is given below.

$$\text{Maximize } Z = \sum_j c_j x_j$$

subject to:

$$\sum_j a_{ij} x_j \leq b_i \; \forall i \tag{35}$$

$$x_j \geq 0 \; \forall j \tag{36}$$

This basic LP formulation of a maximization problem is what Allan utilized.

3.4.4. Algorithm Figure 24 illustrates part of the blocks programming for the simplex app. It is a very efficient implementation.

3.4.5. Demonstration We will demonstrate a simple two dimensional example. The app is capable of two and three dimensional examples with some nice interactive features to change the problems and parameters of the problems.

Let's say that we have the following two-dimensional LP.

$$\begin{aligned} \text{Maximize } Z &= 350x_1 + 120x_2 && (37)\\ s.t. \quad 12x_1 + 5x_2 &\leq 1850 && (38)\\ 8x_1 + 2x_2 &\leq 1200 && (39)\\ x_1, x_2 &\geq 0 && (40)\\ & && (41) \end{aligned}$$

The optimal solution is

$$x_1 = 143.75, x_2 = 25, Z = 5331.$$

As can be seen in the app screens of Figure 25, the Simplex app correctly computes the three simplex tableaus to find the optimal solution. This is quite a feat with AI2.

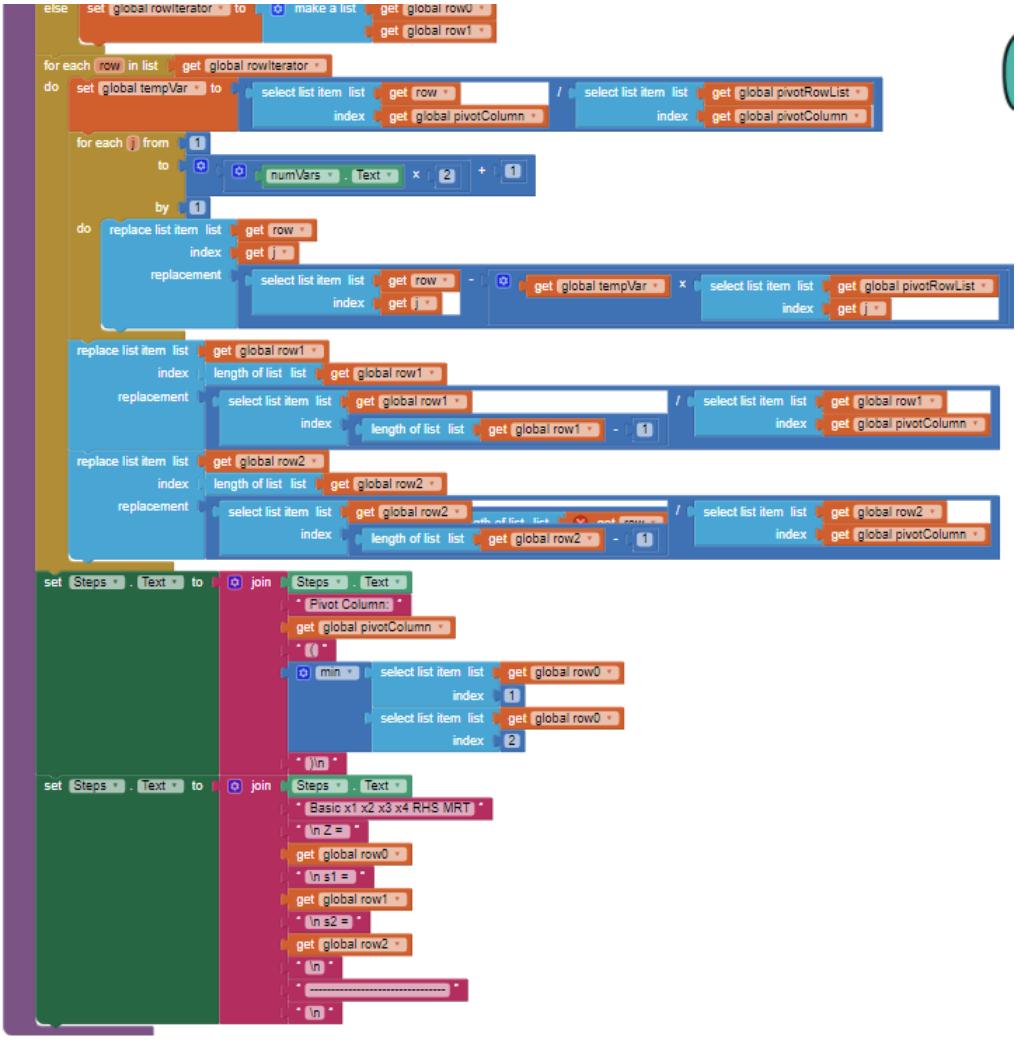


Figure 24. Simplex Algorithm Blocks Programming

3.4.6. Evaluation The app works very well. The printing out of the simplex tableau is an important and useful teaching concept. Now we transition to the next chapter on Integer Programming. Integer programming is an important and challenging research and teaching area. It is described in Chapter 9 of Taha's textbook. It is a very interesting Chapter since Taha does a very good job of illustrating how integer programs are formulated and solved. Integer Programming methods are quite useful although difficult to solve in general since they fall into the realm of $\mathcal{N}P$-complete and $\mathcal{N}P$-Hard problems.

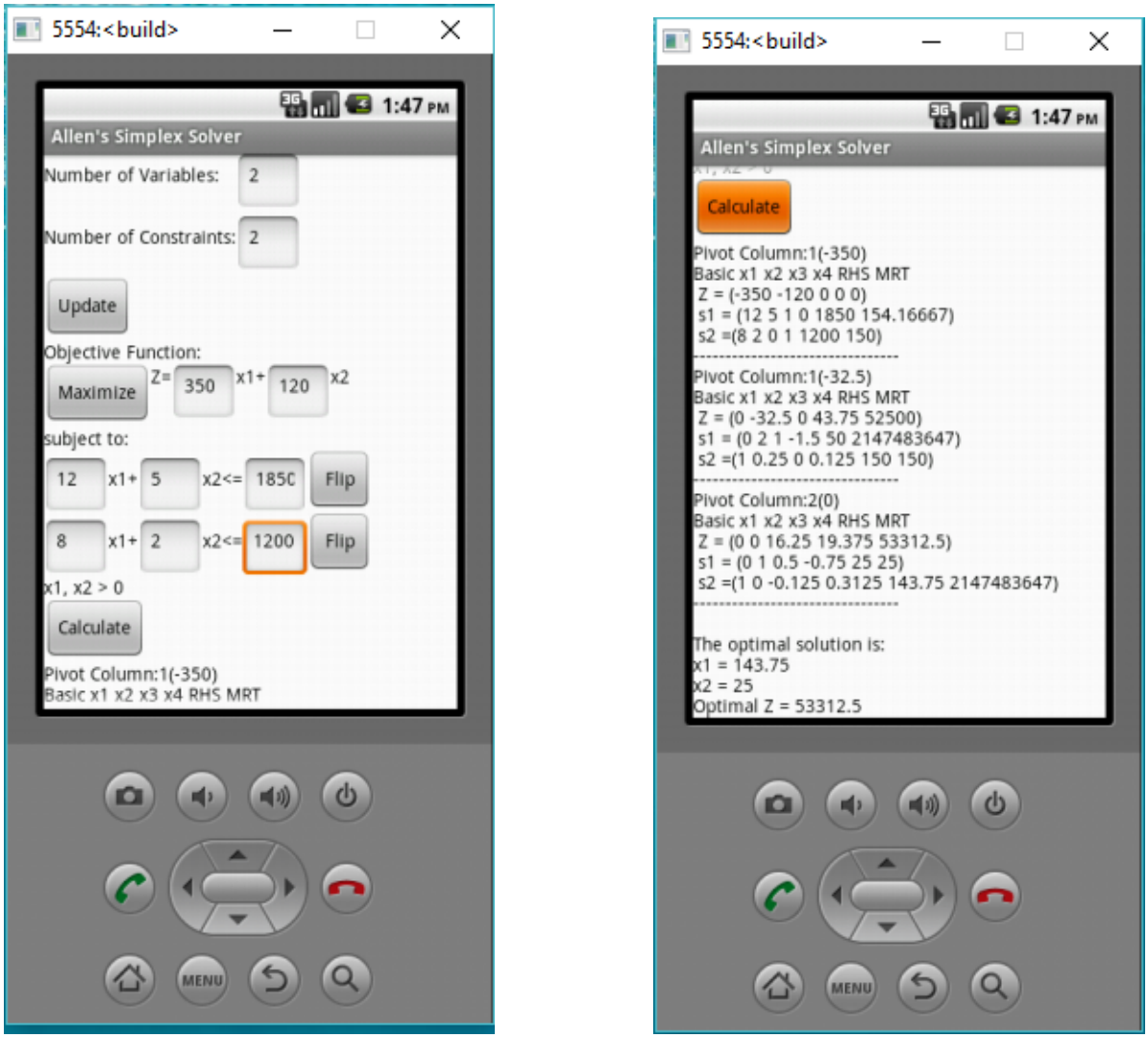


Figure 25. Simplex Algorithm Blocks Programming

4. Integer Programming (IP)

As was discussed, Integer Programming problems have many potential applications since most real world planning and design problems involve integer variables. It is a very rich area and very interesting and there is much research going on for the development of fast algorithms for large scale IPs.

- The general Linear Integer Programming (LIP) problem can be mathematically written as: Find the integer variables $\mathbf{x} = \{x_1, x_2, \ldots x_n\}$ which

$$\text{Maximize } Z = \sum_j c_j x_j$$

subject to:

$$\sum_j a_{ij} x_j \leq b_i \; \forall i \tag{42}$$

$$x_j \geq 0 \; \forall j \tag{43}$$

- The only difference between this problem and the general LP problem is the requirement that x_j be integer, (*i.e.* the divisibility assumption no longer holds).

- The problems where all x_j are required to be integer are called *pure* integer, whereas those problems where only some of the x_j must be integer are called *mixed* integer problems.
- Problems where the x_j are binary variables (*zero-one*) variables are called *binary* integer programs. BIP are very useful in modeling real world applications as we shall see.
- What we will also see here is that there is no simple simplex type algorithm for IPs.

4.1. Swim Team Assignment

The first problem is a classic integer program called the assignment problem. This is a matching/assignment problem which is normally solved with linear programming methods since the properties of the problem allow it to be solved as an integer solution with LP. We actually will utilize the Hungarian assignment method for its solution which is a classical approach for solving this type of problem.

4.1.1. Introduction Suppose the coach of UMass mens team needs to assign swimmers to a 200-yard medley relay team for the next home meet. Since most of the swimmers are very fast in more than one stroke, it is not clear which swimmer should be assigned to each of the four strokes. The four fastest swimmers and the best time (in seconds) appear below in Figure 26.

Events	Al	Ben	Chuck	Dave	Req. # Assignments
Backstroke (E1)	31	33	34	37	1
Breaststroke (E2)	43	33	29	34	1
Butterfly (E3)	33	38	39	28	1
Freestyle (E4)	29	26	30	29	1
# of poss. assignments	$G(N,A)$ 1	$G(N,A)$ 1	$G(N,A)$ 1	$G(N,A)$ 1	4

Figure 26. UMass Swim Times

How should we select the candidates for the relay? There are some obvious choices given the magnitudes of the matrix entities.

4.1.2. Problem Many decision problems involve the *assignment/matching* of elements of one **set** to elements of another set subject to certain rules. Graphically, we can picture the situation as follows in Figure 27:

$$X = \{A, B, C, \ldots, Z\} \rightarrow A := persons, B := time\ slots, C := activities$$

$$Y = \{E_1, E_2, E_3, \ldots, E_m\} \rightarrow E_1 := jobs, E_2 := activities, E_3 := locations$$

Normally, there is also an objective function (performance measure or criterion) that evaluates the value of one particular assignment relative to another, and the problem is to choose which of the possible assignments optimizes the value of the objective function.

- *o* Cost of the assignment ($)
- *o* Preference of each assignment ($1 \ldots 10$; $(-5 \ldots 0 \ldots +5)$)
- *o* Maximum reliability of each assignment ($p \in (0, 1)$)

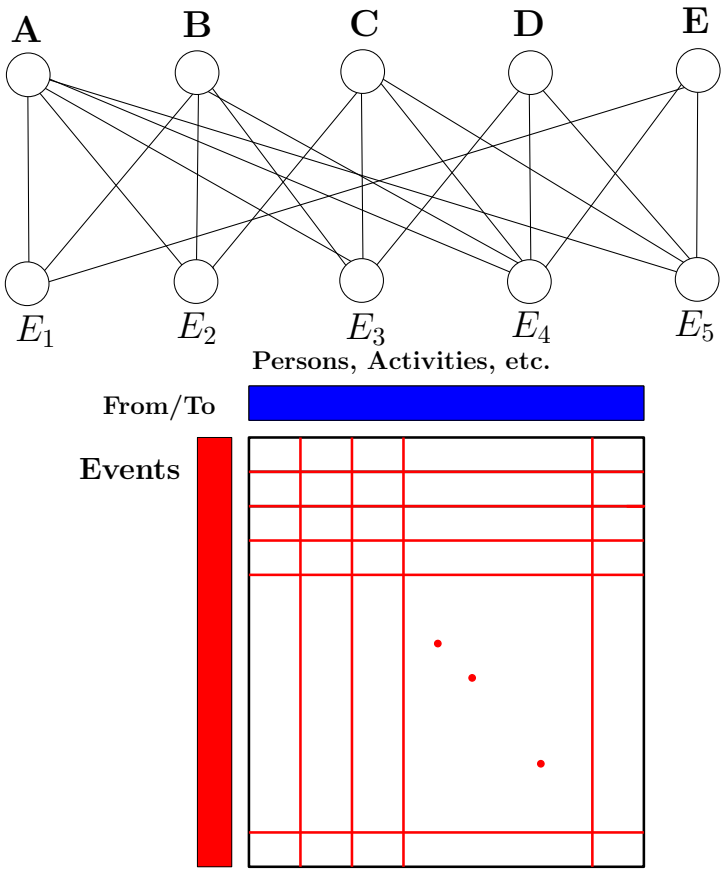
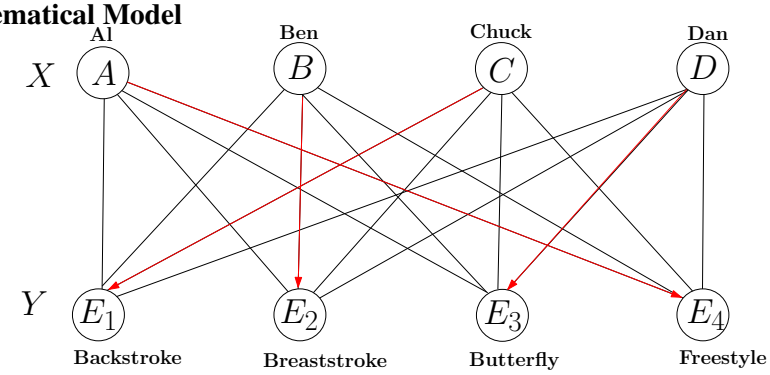


Figure 27. Illustration of Assignment/Matching Problem

4.1.3. Mathematical Model



In the language of graph theory, we have a bipartite graph with vertex set

$$V = X \bigcup Y \qquad \text{and edge set} \qquad E$$

Each edge e connects a vertex of X to a vertex of Y. Moreover $|X| = |Y|$. For each edge $e \in E$ we are given a nonnegative weight w_e. We want to find a subset $M \subseteq E$ of edges such that each vertex of X and Y is incident to exactly one edge of M and the sum $\sum_{e \in M}$ is a minimum.

In order to form this as an optimization problem, let's introduce decision variables x_e, one for each edge of the graph. The decision variables can attain values of either 0 or 1. Thus our problem can be written as:

$$\text{Minimize } Z = \sum_{e \in M} w_e x_e$$

This is our objective function. The requirement that a vertex of $v \in V$ have exactly one incident edge of M is expressed by having the sum of x_e over all edges incident to V equal to one.

$$\sum_{e \in E : v \in V} x_e = 1$$

Thus, we have an Integer Programming Problem (IP):

$$\begin{aligned} \text{Minimize } Z &= \sum_{e \in M} w_e x_e \\ \text{s.t.} \quad & \sum_{e \in E : v \in V} x_e = 1 \\ \text{and } x_e &\in \{0, 1\} \qquad \text{for each edge } e \in E \end{aligned}$$

If we relax the integer requirement on the decision variables, we have:

$$\begin{aligned} \text{Minimize } Z &= \sum_{e \in M} w_e x_e \\ \text{s.t.} \quad & \sum_{e \in E : v \in V} x_e = 1 \\ \text{and } 0 \leq x_e &\leq 1 \qquad \text{for each edge } e \in E \end{aligned}$$

i) This is called an LP relaxation of IP.
ii) What good is this?
iii) It leads to a lower bound on the IP optimum solution. Furthermore, sometimes solving the LP relaxation will afford an optimal solution to the previous problem.
iv) Theorem: If the LP has a feasible solution, the optimal LP solution will be integral.

4.1.4. Algorithm A Modified version of the Hungarian Assignment Algorithm is used to find the optimal assignment. This algorithm is discussed in Chapter of the textbook by Taha. Here is a snapshot of the blocks used in the coding scheme. The AI2 code was developed by Lily Thomas, class of 2012.

4.1.5. Demonstration The app was developed for a high school track team competition rather than a swim meet but obviously the assignment problem is the same in that we have five possible candidates for four different events. Given the input data, the app correctly finds the best assignment for minimizing the total event times.

Figure 29 demonstrates the app for the sample problem.

4.1.6. Evaluation This is a nice compact app and the logic is pretty clear with the blocks programming. Now, let's examine a class of related problems called Transportation problems.

4.2. Transportation Problem

This is also a matching/assignment type problem which has a storied history in Operations Research. Perhaps the first optimization problem ever examined was a Transportation Problem.

4.2.1. Introduction It is very fundamental problem and differs from the matching/assignment model in that the right-hand side resources are not single numbers but represent the integer supplies and demands respectively.

4.2.2. Problem The problem is basically captured in the next diagram. The graph structure is similar to the assignment problem but the difference is the supplies and demands at the nodes are not $1's$. Since the assignment graph structure is similar to the assignment model, the solution to the transportation model will be integer assuming the data are integer.

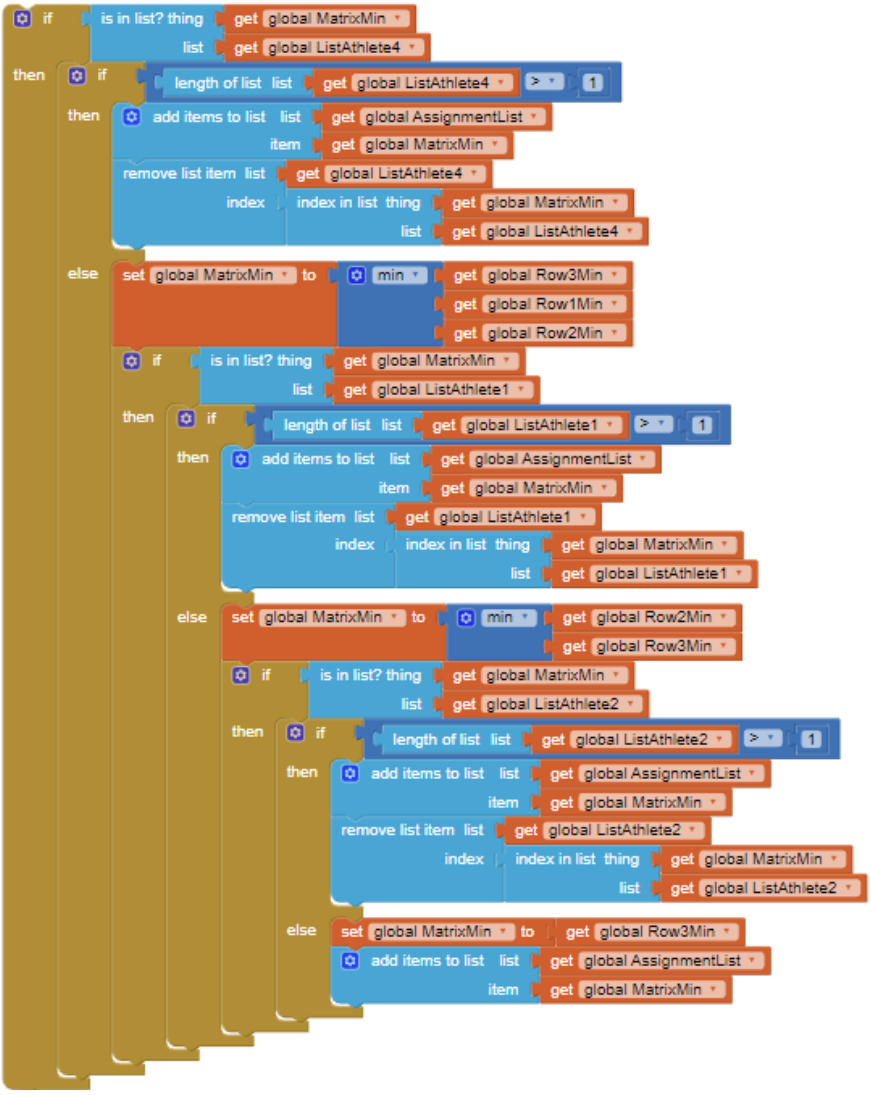
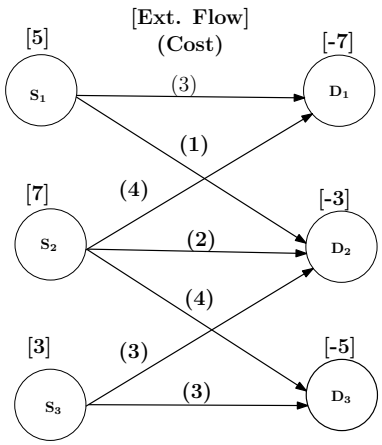


Figure 28. Partial Set of Blocks for the Assignment Algorithm



We see in this example that the supply equals the demand and so this is a balanced transportation problem.

When it is unbalanced, one may have to add dummy surplus sources or dummy demand points in order to balance the problem.

$$\begin{array}{c} \\ S_1 \\ S_2 \\ S_3 \\ Demand \end{array} \begin{array}{c} \begin{array}{cccc} D_1 & D_2 & D_3 & Supply \end{array} \\ \begin{bmatrix} 3 & 1 & M & 5 \\ 4 & 2 & 4 & 7 \\ M & 3 & 3 & 3 \\ 7 & 3 & 5 & 15 \end{bmatrix} \end{array}$$

4.2.3. Mathematical Model In general, a transportation problem is specified by the following information:

Figure 29. Solution App for the Assignment Algorithm

1. A set of **m** supply points from which a good is shipped. Supply point i can supply at most a_i units.
2. A set of **n** demand points to which a unit good is shipped. Demand point j must receive at least b_j units of the shipped goods.
3. Each unit shipped incurs a transportation cost c_{ij}.

- Let $\mathbf{x_{ij}}$ be the number of units shipped from supply point i to demand point j. Then the general formulation of the transportation problem is:

$$\text{Min} \sum_i \sum_j c_{ij} x_{ij}$$

subject to:

$$\sum_j x_{ij} \leq a_i \; \forall \; i \text{ supply points} \tag{44}$$

$$\sum_i x_{ij} \geq b_j \; \forall \; j \text{ demand points} \tag{45}$$

$$x_{ij} \geq 0 \; \forall \; (i,j) \text{ pairs} \tag{46}$$

4.2.4. Algorithm The general transportation problem with m sources and n destinations has $m+n$ constraint equations, one for each source and each destination. However, because the transportation model is always balanced (sum of supply = sum of demand), one of these equations is redundant. Thus, the model has $m+n-1$ independent constraint equations, which means that the starting basic solution consists of $m+n-1$ basic variables. We must insure that in the starting solution, we have $m+n-1$ basic variables. The special structure of the transportation problem allows securing a non-artificial starting bfs using one of the three methods:

1. Northwest-corner method.
2. Least-cost method.
3. Vogel approximation method.

The three methods differ in the quality of the starting bfs so as to minimize the eventual number of pivots to reach the optimal solution. The VAM Method is the one we will implement in the phone apps.

Step 1. For each row (column), determine a penalty measure by subtracting the *smallest* unit cost element in the row (column) from the *next smallest* unit cost element in the same row(column).

Step 2. Identify the row or column with the largest penalty. Break ties arbitrarily. Allocate as much as possible with the least unit cost in the selected row (column). Adjust the supply and demand, and cross out the satisfied row or column. If a row or column are satisfied simultaneously, only one of the two is crossed out, and the remaining row(column) is assigned zero supply (demand).

Step 3.

(a) If exactly one row or column with zero supply or demand remains uncrossed, stop.
(b) If one row(column) with *positive* supply (Demand) remains uncrossed out, determine the basic variables in the row(column) by the least-cost method, and stop.
(c) If all the uncrossed out row and columns have (remaining) zero supply and demand, determine the *zero* basic variables by the least cost method. Stop.
(d) Otherwise, go to Step 1.

4.2.5. Demonstration

4.2.6. Evaluation

4.3. Zero-One Project Scheduling

This is a classical example of an Integer Programming Project Scheduling problem. We have a list of project activities we need to schedule and we have a list of times for each project activity and certain due dates that need to be fulfilled. We are trying to schedule these activities for ourselves so there is only one processor or decision maker.

4.3.1. Introduction In most personal and professional settings, everyone needs to determine which activities to perform and how to schedule them over time. Being able to prioritize which projects should be completed and in the right order can be most challenging when due dates, importance, and the time required to complete each task varies.

4.3.2. Problem We would like to complete a set of activities over time when there a re due dates, times for the activities, and preference importance are known. The app will create an estimated start date for each project activity. The overall objective is to minimize the lateness of the entire set of activities. The app was coded by Joe Woodman of the class of 2015.

4.3.3. Mathematical Model The problem becomes a mixed integer linear programming problem. The decision variables are:

- x_j:= the Start date for job j (measured from time zero)
-

$$y_{ij} = \begin{cases} 1 & \text{if } i \text{ precedes } j, \\ 0, & \text{if } j \text{ precedes } i. \end{cases} \tag{47}$$

The problem has two types of constraints: the noninterference constraints (guaranteeing that no two jobs are processed concurrently) and the due-date constraints. Two jobs i and j with processing time p_i and p_j will not be processed concurrently if (depending on whether which job is processed first)

$$x_i \geq x_j + p_j \text{ or } x_j \geq x_i + p_i \tag{48}$$

For M sufficiently large the "or" constraints are converted to "and" constraints by using

$$My_{ij} + (x_i - x_j) \geq p_j \text{ and } M(1 - y_{ij}) + (x_j - x_i) \geq p_i \tag{49}$$

The conversion guarantees that only one of the two constraints will be active at any one time.

Next, given that d_j is the due date for job j, the job is late if $x_j + p_j > d_j$. We use two nonnegative variables x_j^- and s_j^+ to determine the status of a completed job j with regard to its due date. The due date constraint is written as:

$$x_j + p_j + s_j^- - s_j^+ = d_j \tag{50}$$

Job J is ahead of schedule if $s_j^- > 0$ and late if $s_j^+ > 0$. The mathematical model is then :

$$\text{Minimize } Z = \sum w_j s_j^- \tag{51}$$
$$My_{ij} + (x_i - x_j) \geq p_j \tag{52}$$
$$M(1 - y_{ij}) + (x_j - x_i) \geq p_i \tag{53}$$
$$x_j + p_j + s_j^- - s_j^+ = d_j \tag{54}$$
$$x_j, s_j^-, s_j^+ \geq 0 \; \forall j \tag{55}$$
$$y_{ij} = (0, 1) \forall (i, j) \tag{56}$$

Please see Taha, Chapter 9 for more details about this type of job scheduling model.

4.3.4. Algorithm The underlying strategy for solving this problem is a Branch & Bound approach which is a classical method for solving IP problems. Linear Programming upper bounds are normally solved for at the beginning stages of the search process then lower bounds are estimated to prune the search tree. The lower bounds are solved with Dual Simplex LP iterations. This is also standard practice.

Figure 30 shows a partial sample of the blocks programming code. This program is solved for on the NEOS server as the complexity of the approach is beyond the scope of AI2.

4.3.5. Demonstration Figure 31 illustrates the process of inputting the data for the example demonstration where screen one on the left shows the start of the project and screen two on the right shows the data requirements for each project activity which must be input. Once all the project activity data are complete, one sends the app to the NEOS server for the solution.

The final solution from the NEOS server is:

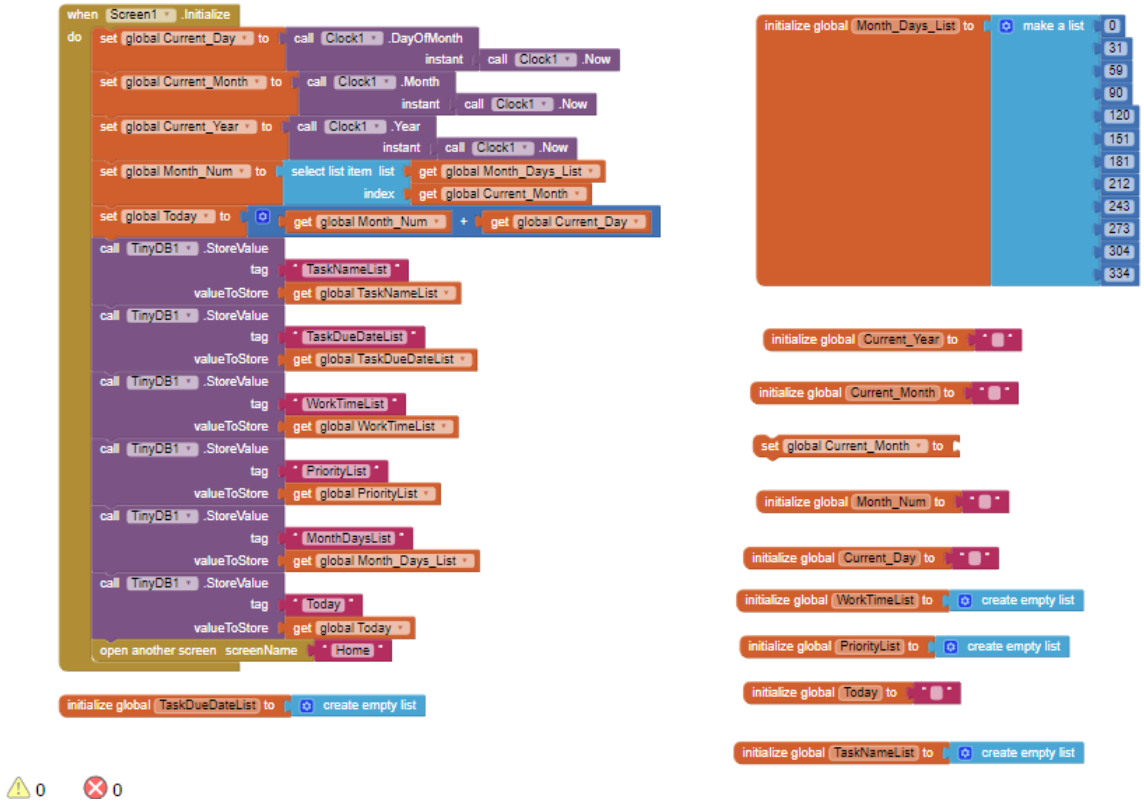


Figure 30. Project Scheduling Blocks

```
Presolve eliminates 0 constraints and 4 variables.
Adjusted problem:
24 variables:
        12 binary variables
        12 linear variables
28 constraints, all linear; 84 nonzeros
        4 equality constraints
        24 inequality constraints
1 linear objective; 4 nonzeros.

CPLEX 12.7.0.0: threads=4
CPLEX 12.7.0.0: optimal integer solution; objective 2.2312
42 MIP simplex iterations        0 branch-and-bound nodes
No basis.
penalty Z = 2.2312
x [*] :=  x1=2.24   x2=  0    x3=  1.24    x4=  0.62
```

4.3.6. Evaluation The user interface is very good on this app. It is very easy to enter the project data and the date app is very convenient.

4.4. p-Median and p-center Location Problems

Special network location problems with integer variables but can be solved with clever network enumeration approaches.

Figure 31. Project Scheduling App Demo

4.4.1. Introduction

4.4.2. Problem

4.4.3. Mathematical Model

4.4.4. Algorithm

4.4.5. Demonstration

4.4.6. Evaluation

4.5. Knapsack Problems

Integer Programming problem applications with special structure.

4.5.1. Introduction

4.5.2. Problem

4.5.3. Mathematical Model

4.5.4. Algorithm

4.5.5. Demonstration

4.5.6. Evaluation

4.6. Set Covering and Set Packing

Another Integer Programming problem with special structure. Chapter 9 in Taha discusses these problems.

4.6.1. Introduction

4.6.2. Problem

4.6.3. Mathematical Model

4.6.4. Algorithm

4.6.5. Demonstration

4.6.6. Evaluation

5. Nonlinear Programming (NLP)

Nonlinear programming problems tend to be some of the most challenging problems we face in optimization. Chapters 20 and 21 of Taha discuss the general nature of NLP problems. There are many, many applications for NLP type problems. The general structure is the following programming problem.

$$\text{Max or Min } f(\mathbf{x})$$

subject to:

$$\text{inequalities } g_i(\mathbf{x}) \leq 0 \tag{57}$$
$$\text{equalities } h_i(\mathbf{x}) = 0 \tag{58}$$
$$\text{nonnegativity } \mathbf{x} \geq 0 \tag{59}$$

The form of these functions can be quite complex. If the objective functions and constraints are convex, then it might be possible to get global optimal solutions, but in general, one must be happy with local optimal solutions because of the computational difficulty of these problems.

Nonlinear Programming methods are quite vast and are appropriate for unconstrained, equality constrained, and inequality constrained problem formulations. We will examine a number of these nonlinear problems since they have many real world applications, especially for engineering design applications. In some of these example apps, the use of calculus affords us directly the equations for optimization. The equations can be nonlinear, but if the expressions are not too complex, AI2 can be utilized.

5.1. Gas Guzzler

This is a problem of trying to find the optimal speed with which to drive a car to minimize the cost of driving a car in relation to the maintenance and operation of the car. We shall use calculus to solve this unconstrained optimization problem.

5.1.1. Introduction Driving and minimizing the cost of gasoline while driving is an important task most people cherish because of the unpredictable gas prices. The idea for this app problem first appeared in a textbook on optimization by Russel [4].

5.1.2. Problem Let's say we wish to plan a long-distance trip and we wish to determine the optimal speed decision variable s with which we should drive during the duration of the trip, which has a given distance or contextual variable d. Further, say the trip is between Boston and Chicago. We are concerned about our gas mileage because it has become so expensive to drive in the last few years. We want to first minimize $f_1(s)$ the costs of operating our car and second minimize the mileage costs $f_2(s)$ and optimize our speed decision variable s to offset the rising cost of gasoline. Thus, the total cost of our trip is:

$$f(\text{total cost}) = f_1(s)(\text{operating \$}) + f_2(s)(\text{mileage\$})$$

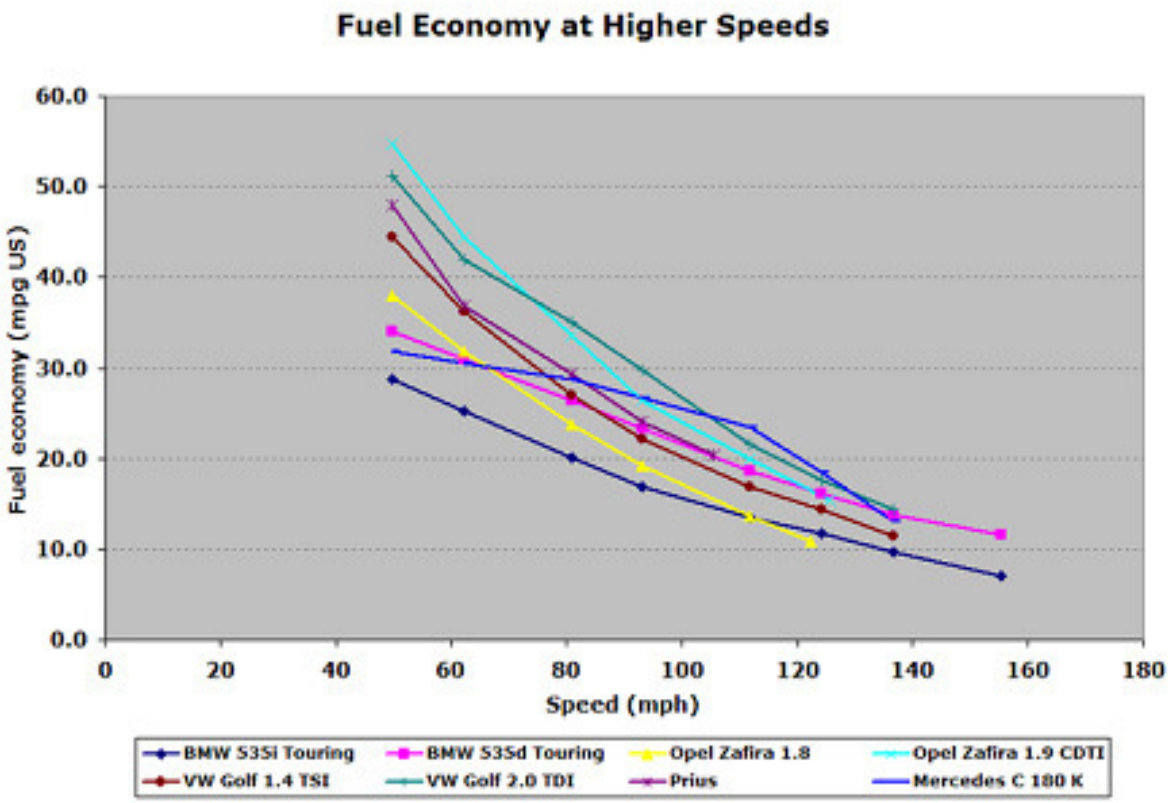
There are no explicit constraints in the problem which makes it an unconstrained optimization problem. Unconstrained optimization problems are easier to solve than constrained problems.

One can show empirically that mileage per gallon m has the following relationship to speed s

$$m = v - \frac{s}{c}$$

- where m:= mileage per gallon

- v:= y-intercept of mileage speed cure
- c:= slope factor in the mileage speed curve.
- h:= operating costs of the car ($)
- g:= cost of gas per gallon ($)



From a study of the Federal Government which shows the gas mileage for different car-types on the y-axis and the speed of the vehicle on the x-axis. v in our model is the linear equation y-intercept.

5.1.3. Mathematical Model

- Further, from our operating cost and mileage cost expressions:

$$f_1 = \frac{hd}{s} \rightarrow \frac{\$/\text{hr} * \text{miles}}{\text{miles/hr}}$$

$$f_2 = \frac{gd}{m} = \frac{gd}{v - \frac{s}{c}} \rightarrow \frac{\$/\text{gallon} * \text{miles}}{\text{miles/gallon}}$$

$$f \text{ totalcost } = \frac{hd}{s} + \frac{gd}{v - \frac{s}{c}}$$

- If we differentiate the total cost function f and set to 0, we get:

$$f'(s) = -\frac{hd}{s^2} + \frac{gd}{(v - \frac{s}{c})^2 c} = 0$$

- This is a quadratic equation, we can solve for s with the quadratic formula, since it will have two roots:

$$(a_1, a_2) \rightarrow a_1 = \frac{(-2h + 2\sqrt{hgc})cv}{2(-h + gc)}; a_2 = \frac{(-2h - 2\sqrt{hgc})cv}{2(-h + gc)}$$

5.1.4. Algorithm The AI2 blocks for the Gas Guzzler model are depicted in Figure 32. Again, because of the calculas equations is is pretty straightforward to program.

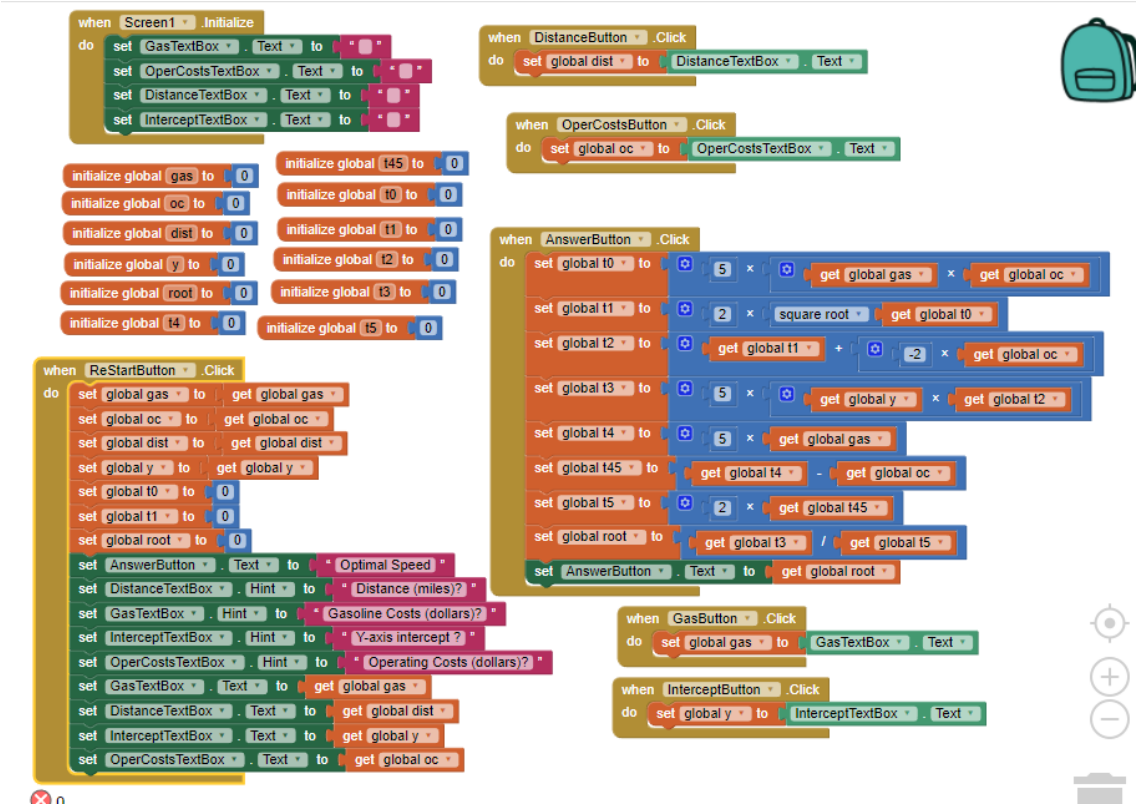


Figure 32. Gas Guzzler Blocks

5.1.5. Demonstration Figure 34 on the left illustrates the designer screen of the Gas Guzzler app. Figure 33 illustrates the actual objective function curve for our example problem.

For example, let's say that we have the following parameters for our example problem:

$h = \$1$ operating costs/hour

$d = 1000$ miles;

$c := 5$ a constant;

$g := \$3.5/gal$;

$v\ (\mathrm{y} - \mathrm{intercept}) := 60$

Then solving for s, we get two real roots: $57.878, -94.242$ Obviously, the first positive root is our solution as is indicated in Figure 34. The convexity of the objective function is an interesting property of the problem as shown in Figure 33.

5.1.6. Evaluation This is a nice simple app with a straightforward solution. Of course coming up with the nonlinear equations was the real trick.

5.2. Price is Right

This is an interesting problem from Micro Economics and we shall also use calculus to solve this problem. We have a demand curve and we seek to find the optimal price of an item subject to its demand and un it cost. This app shows the nice relationship between linear quantities and nonlinear quadratic functions.

5.2.1. Introduction To demonstrate a simple model using the quantities price, demand and unit cost and their interrelationships.

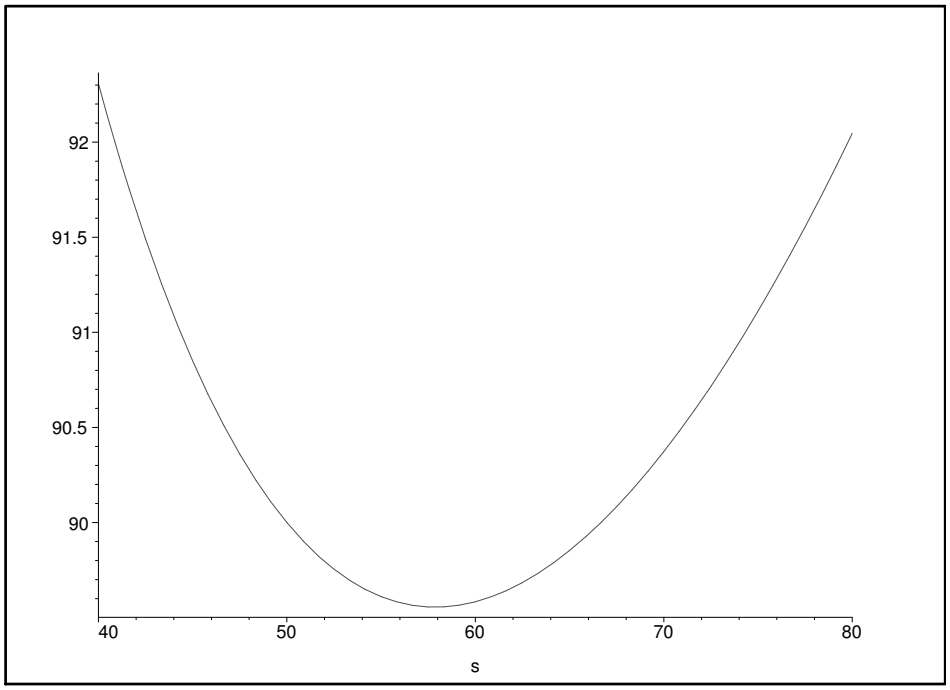


Figure 33. Gas Guzzler App

5.2.2. Problem A firm's production department has found that the unit cost of its main product is c dollars. Meanwhile, the marketing department has estimated the relationship between the slope (p) and the maximum demand for the product maximum demand (D) (sales volume) follows the linear "demand curve", so that y the demand is a linear function of the price x. We need the following notation:

Variables	Description
$c :=$	unit cost
$D; =$	Maximum Demand
$p :=$	Slope of demand curve
$x :=$	Selling price per unit
$y :=$	Unit demand
$z :=$	Overall objective function value

The demand is a linear function of the price with D intercept and slope p.

$$y = D - px \tag{60}$$

Figure 35 illustrates the linear relationship of price and demand.

5.2.3. Mathematical Model The decision variable x is the selling price/unit in dollars. We would like to maximize our profit for the firm where we have the nonlinear relationship given by the following equation.

$$z = (x - c)y \tag{61}$$

Substituting Equation 60 for y and multiplying through with 61, we get

$$z = (x - c)(D - px) \tag{62}$$

or re-writing this we get a quadratic function Equation 63:

Figure 34. Optimal Mileage Estimate

$$z = -px^2 + xD - Dc + pcx \tag{63}$$

If we differentiate this expression with respect to x and set it equal to 0:

$$z' = -2px + D + cp = 0 \tag{64}$$

and solving for x, we get:

$$x^* = \frac{D + cp}{2p} \tag{65}$$

Equation 65 represents the optimal price of the unit. This latter equation is the key to the entire app.

5.2.4. Algorithm Figure 36 shows the complete set of blocks that are needed for this application. Because this is not a very complex algorithm, it is straightforward to program it.

Below are two clickable hyper links (place them in a browser separately) to show how the app was built. The video demonstration is in two parts with the first part describing the building of the designer screen and the second part showing how the visual programming blocks were assembled.

udrive.oit.umass.edu/jmgsmith/MIE%20379%20PriceRightDesignComponents/MIE%20379%20PriceRightDesignComponents.ht

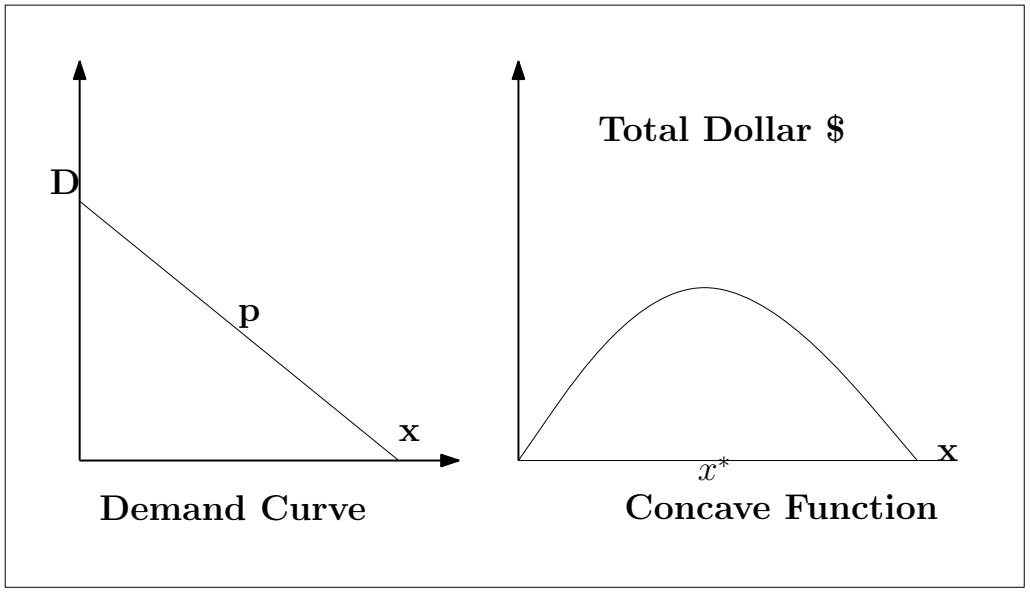


Figure 35. Price Demand Curve Relationships

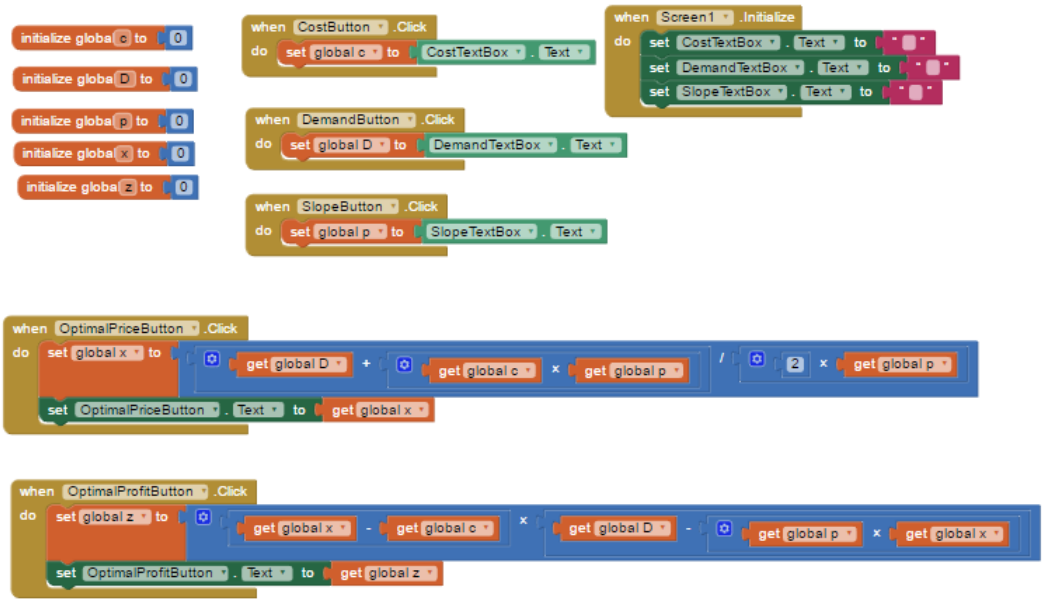


Figure 36. Complete Blocks of the Price-is-right App

udrive.oit.umass.edu/jmgsmith/MIE%20379%20PriceBlocks/MIE%20379%20PriceBlocks.html

5.2.5. Demonstration Figure 37 illustrates the home screen and the final solution for an example problem.

5.2.6. Evaluation The price is right app is very simple, but that is because we have used the calculus to find the key equations. The next app is a little more ambitious and shows how we can utilize App Inventor in an iterative nonlinear environment.

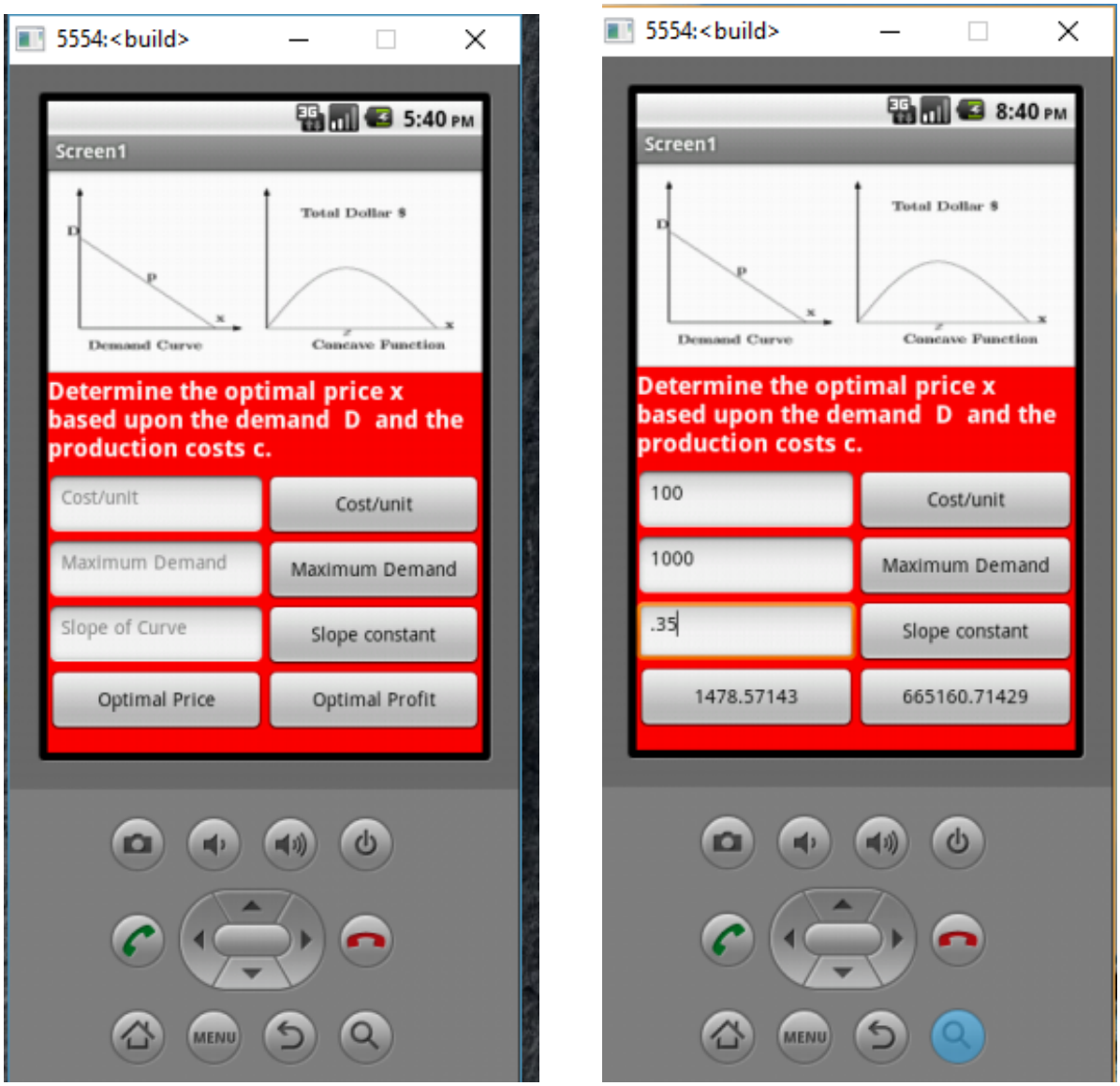


Figure 37. Price is Right App

5.3. Weber Location Problem: Weizfeld's Algorithm

This is a classical nonlinear location problem with Euclidean distance. We want to demonstrate a dynamic location problem where the inputs are put in by the user and the app computes the optimal location of a new facility after the inputs. Also, of some import, the app demonstrates how multiple lists are searched in App Inventor 2.

5.3.1. Introduction Let's say we are working out of doors and need to determine the location of a new well in relation to some existing wells. The phone app environment is most suitable for this type of situation. We have a map of the area and we want to locate the existing well with Cartesian coordinates and then determine the new well in relation to the existing coordinates. As in the previous factory location problem, we have a given set of demand points and we are interested in locating a new facility in relation to the given facilities such that the unweighted distance from the new facility to the existing facilities is minimized.

5.3.2. Problem The classical version of the problem is now defined. Given three points in the plane, find a fourth point $s : (X, Y)$ such that the sum of its distances to the other three points is a minimum (Fermat, 17th century):

$$Minimize\ Z = f(\mathbf{x}) = \sum_{i=1}^{3} w_i d_2(i, j)$$

where $w_i =$ flow from points i to points j

$$d_2(i, j) = [(X - a_i)^2 + (Y - b_i)^2]^{1/2}$$

P_i has (a_i, b_i) coordinates. The above is an "unconstrained" nonlinear programming problem, often referred to as the Steiner/Weber problem.

We actually have already seen this problem once before in the Pinball Weber Location app, but in this instance, we will develop a nonlinear programming algorithm approach. In this instance, Euclidean distance will be followed rather than rectilinear distance.

5.3.3. Mathematical Model There can actually be more than three given facilities and our app will allow for this. We assume Euclidean distance is used to travel between the new facility $i(x, y)$ and the $j-$existing facilities located at points (a_i, b_i)

$$\text{Minimize Z } = \sum_{j} w_j dist(i, j)$$

So we will also have an unconstrained optimization problem in two variables (X, Y). Classical Calculus:

$$\frac{\partial Z}{\partial X} = \frac{\partial Z}{\partial Y} = 0$$

the above are necessary conditions for the optimality of point s. If $f(\mathbf{x})$ is convex, then the above conditions are also sufficient to guarantee optimality.

$f(\mathbf{x})$, however, is not separable in X, Y.

As mentioned before, it is difficult to directly solve $\frac{\partial Z}{\partial X}, \frac{\partial Z}{\partial Y}$ because they are interdependent nonlinear equations and the square roots create round-off error as well as computation problems.

5.3.4. Algorithm The algorithm we will follow is called Weiszfeld's algorithm. Partial derivatives are given by the following set of equations:

$$\begin{aligned}
\frac{\partial Z}{\partial X} &= \sum_{i=1}^{m} \frac{w_i(X - a_i)}{[(X - a_i)^2 + (Y - b_i)^2]^{\frac{1}{2}}} = 0 \\
\frac{\partial Z}{\partial Y} &= \sum_{i=1}^{m} \frac{w_i(Y - b_i)}{[(X - a_i)^2 + (Y - b_i)^2]^{\frac{1}{2}}} = 0
\end{aligned} \tag{66}$$

If we set the first equation to zero and perform some algebra:

$$\begin{aligned}
X \sum_{i=1}^{m} \frac{w_i}{[(X - a_i)^2 + (Y - b_i)^2]^{\frac{1}{2}}} = \\
\sum_{i=1}^{m} \frac{w_i a_i}{[(X - a_i)^2 + (Y - b_i)^2]^{\frac{1}{2}}}
\end{aligned} \tag{67}$$

If we further let:

$$g_i(X,Y) = \frac{w_i}{[(X-a_i)^2 + (Y-b_i)^2]^{\frac{1}{2}}} \forall i \tag{68}$$

We get the following:

$$X = \frac{\sum_{i=1}^{m} a_i g_i(X,Y)}{\sum_{i=1}^{m} g_i(X,Y)} \tag{69}$$

and for the Y-coordinate:

$$Y = \frac{\sum_{i=1}^{m} b_i g_i(X,Y)}{\sum_{i=1}^{m} g_i(X,Y)} \tag{70}$$

Thus, if we start from a given point (usually the center of gravity) then we can compute the next point iteratively as:

$$X^{k+1} = \frac{\sum_{i=1}^{m} a_i g_i(X^k,Y^k)}{\sum_{i=1}^{m} g_i(X^k,Y^k)} \tag{71}$$

and for the Y-coordinate:

$$Y^{k+1} = \frac{\sum_{i=1}^{m} b_i g_i(X^k,Y^k)}{\sum_{i=1}^{m} g_i(X^k,Y^k)} \tag{72}$$

This is referred to as Weizfeld's algorithm (1939).

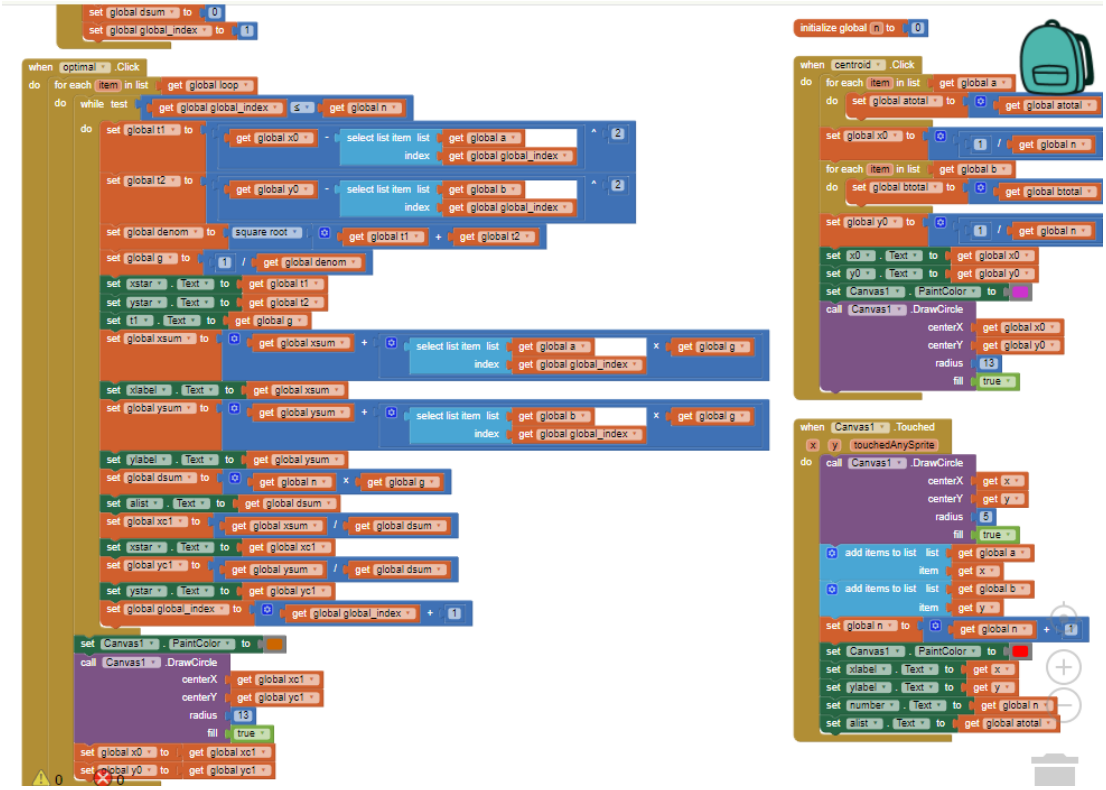


Figure 38. Partial Blocks of the Weiszfeld Location App

5.3.5. Demonstration The designer screen has the origin in the upper left hand corner. The USGS map is an area in part of eastern Massachusetts. The user inputs the apps by hand on the screen and the app records the (x, y) existing coordinates as they are entered. Figure 39 illustrates the input and final optimal solution for an example 3-point problem. The purple circle is the starting solution while the orange circle is the final optimal solution after five iterations. More iterations are possible, but five was a test run for the app. Re-setting the app will run another problem input by the user.

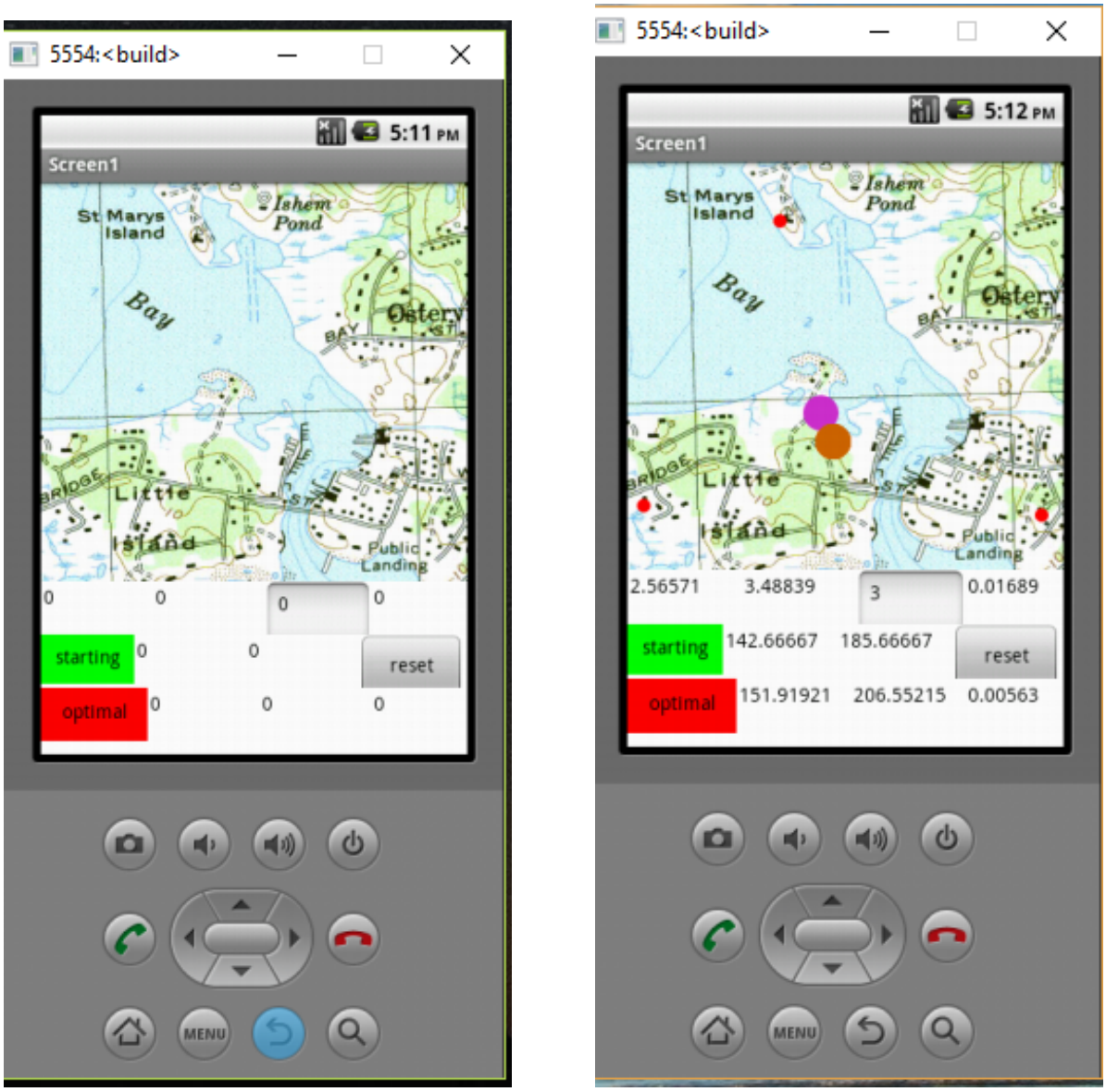


Figure 39. Weiszfeld Location App

5.3.6. Evaluation The program is pretty efficient and seems to work well. Another related problem is the smallest enclosing sphere problem which is treated in the next app.

5.4. Smallest Enclosing Sphere Problem

A classical geometric location optimization problem is the minimum enclosing circle or sphere problem. There are a number of different algorithms for its solution and it maintains an intuitively appealing understanding.

5.4.1. Introduction There are many practical applications.

o Find the location of a hospital or post office to service a community population so that the largest distance to the facility is minimized,

o In the military, this is known as the "smallest bomb problem" for obvious reasons.

o Jung developed a theorem which states that every finite set of points with geometric span has an enclosing circle with radius no greater than $\frac{d}{\sqrt{3}}$. So the problem has a solution, in fact, there is an optimal solution.

5.4.2. Problem This is a classical optimization problem *a.k.a* as the *Minimum Enclosing Sphere (Ball) problem*. We are given a set of randomly generated set of points S in the plane/space and we wish to find the ball(circle) B of smallest diameter such that all points in S are or higher dimensions either contained in B or are on its boundary. It has many applications:

Figure 40 illustrates three alternative spheres enclosing a common point set. We would like to find the smallest enclosing sphere.

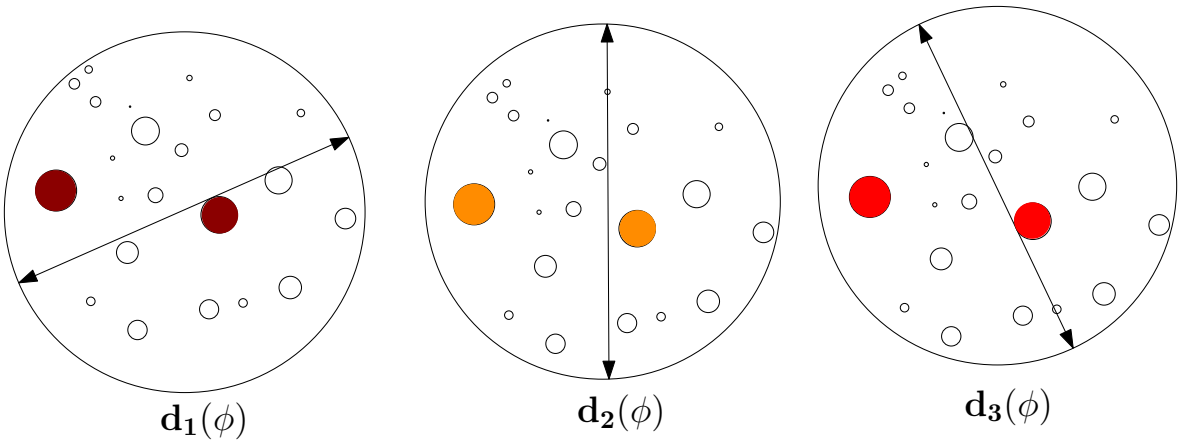


Figure 40. Smallest Enclosing Sphere Alternatives

Wendy (Xi Jiang) from Smith College who took our class in 2015 programmed the app. She was interested in finding the best party location for four friends in the Pioneer Valley.

5.4.3. Mathematical Model Let the coordinates of the existing points P_i be denoted as $(a_i, b_i) i = 1, \ldots, 4$ and those of the desired point P be given as (x, y). Define $d(x, y) = \max(d(P, P_i)), i = 1, \ldots 4$. Then $d(x, y) \geq d(P, P_i)$, equivalently

$$d(x, y)^2 \geq (x - a_i)^2 + (y - b_i)^2, i = 1, 2, 3, 4.$$

By defining a new variable $\lambda = x^2 + y^2 - d^2$, the problem is reduced to the following quadratic programming problem:

$$\text{Minimize } f(\lambda, x, y) = x^2 + y^2 - \lambda \tag{73}$$
$$\text{s.t.} \quad 2a_i x + 2b_i y - \lambda \geq a_i^2 + b_i^2, i = 1, 2, 3, 4 \tag{74}$$

This quadratic programming problem can then be solved on the NEOS server.

5.4.4. Algorithm

5.4.5. Demonstration Each of the alternative possible locations is encoded in a list structure within the app with their latitude and longtitude locations. These locations are then sent to the Neos server along with the optimization problem formulation and eventually solved by the software code Minos. Figure 42 illustrates the simple interface and the alternative locations possible.

The message received from the Neos server on the phone or tablet yields the optimal location for the meeting point as shown below.

Figure 41. Smallest Enclosing Sphere App

5.4.6. Evaluation This is a very nicely designed and compact app. It is a complex nonlinear programming problem but it works very well with the NEOS server. The user input is controlled by the sample list of locations, but this is eminently workable and demonstrates that more locations or interactively inputting the coordinates is all possible,

5.5. Disc Brake Engineering Design

This is an example of an engineering design project where a number of pieces of data must be brought together to optimize an engineering system. Normally, these projects require a set of nonlinear equations to be solved so that is why this project in in the NLP category.

5.5.1. Introduction Multiple disk brakes can be used to deliver extremely high torque in minimal dimensional requirements. The multiplication of surface area in a multiple disk brake allows for one of the smallest torque to size ratios available. Several factors such as inner radius, outer radius and brake torque have to be determined. The app is designed to solve for those quantities.

5.5.2. Problem A multiple disk brake is driven by a Direct drive housed brushless motor RBE and the motor type can be selected from a list picker in the app. The app contains a database of motor types which the user selects from a drop down list. The end of the brake is connected to a rack and pinion which acts as the system output. The output system is 1.75 of the motor weight and the friction coefficient is assumed to be 0.20 Meanwhile, the force is transmitted hydraulically through a piston. The inner and outer radius of this brake as well as the brake torque and required force are to be determined with the app.

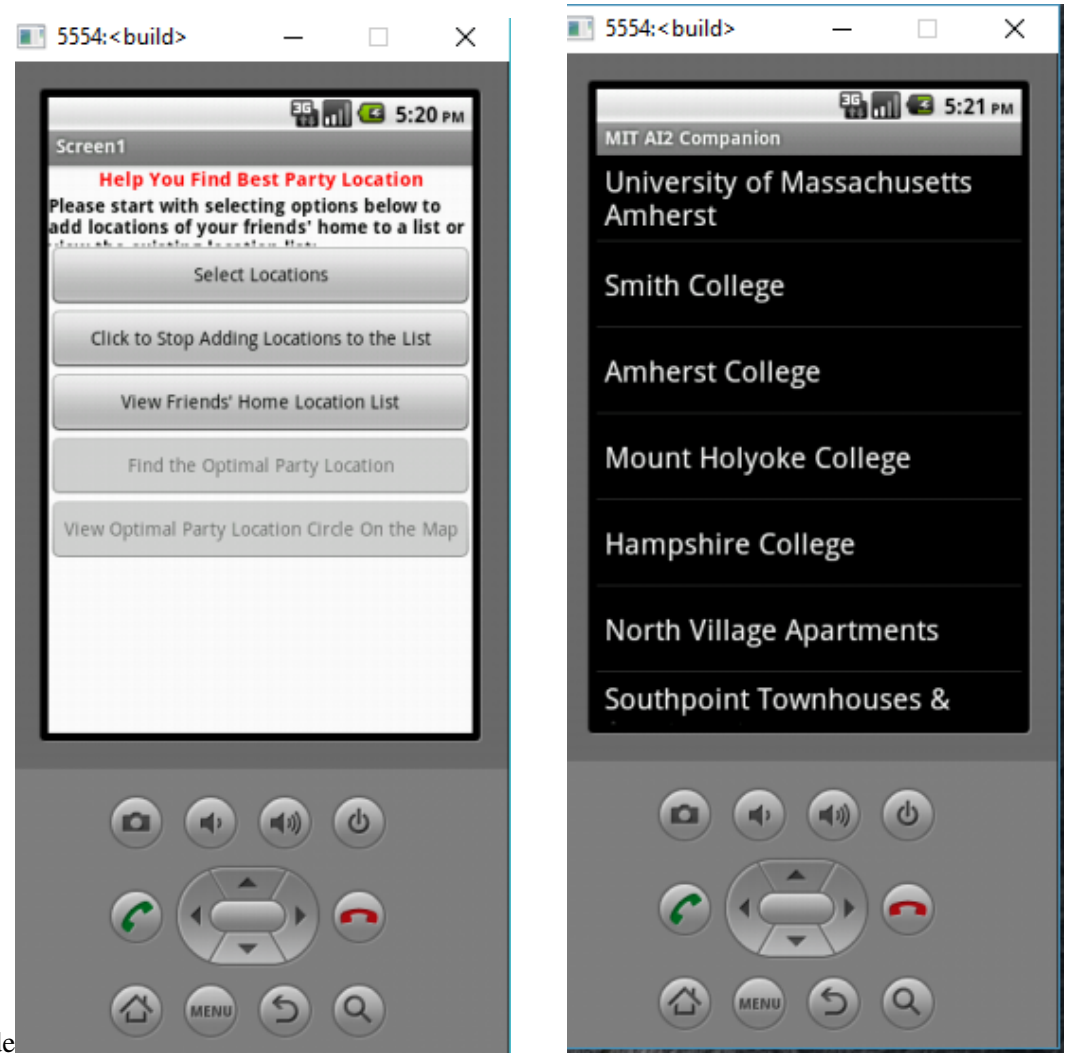
leavevmode 

Figure 42. Smallest Enclosing Sphere App Demo

5.5.3. Mathematical Model Figure 44 illustrates the disk brake system relationships.

The forces involved in the disk brake system which are important for the phone app are the following:

Total normal force acting on the brake:

$$F = \int 2\pi p r dr = \pi p(r_0^2 - r_i^2)$$

Brake Torque:

$$T = \int_{r_i}^{r_0} 2\pi p_{max} r_i N F_r dr = \pi p_{max} r_i f(r_0^2 - r_i^2) N$$

Torque, force and outside radius relations:

$$F = T/(f 1.58 r_0 N)$$

5.5.4. Algorithm Figure 45 lists a sample of the blocks for the disk brake optimization. There are several lists of input data crucial to the app optimization. Multiple screens are used to help with the input data process.

```
neos@neos-server.org
11:44 AM (1 hour ago)
 to me
File exists
You are using the solver minos.
Executing AMPL.
processing data.
processing commands.
Executing on prod-exec-5.neos-server.org
3 variables:
 2 nonlinear variables
 1 linear variable
4 constraints, all linear; 12 nonzeros
4 inequality constraints
1 nonlinear objective; 3 nonzeros.

MINOS 5.51: optimal solution found.
3 iterations, objective 1766.752666<-------
Nonlin evals: obj = 6, grad = 5.
x = 21.1591 <-----------------------------
y = 36.3186 <-----------------------------
```

Figure 43. Neos Return Message with Optimal Location Coordinates

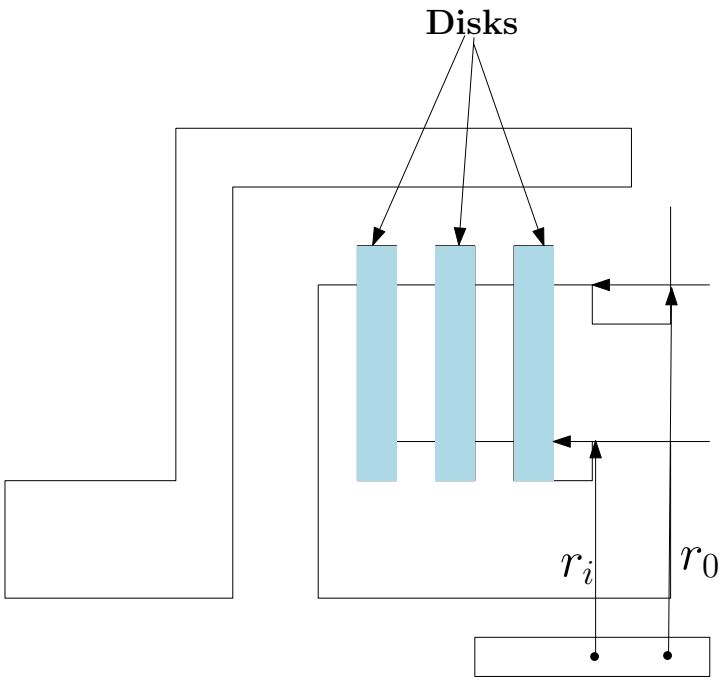


Figure 44. Disk Brake System Diagram

5.5.5. Demonstration Figure 46 illustrates the input process for the brake design and the solution for the Torque and the radius decision variables for the particular motor type. Once the motor type is specified one has to input the Friction Surface and then the Brake Outer Radius, then the app will optimize the remaining values.

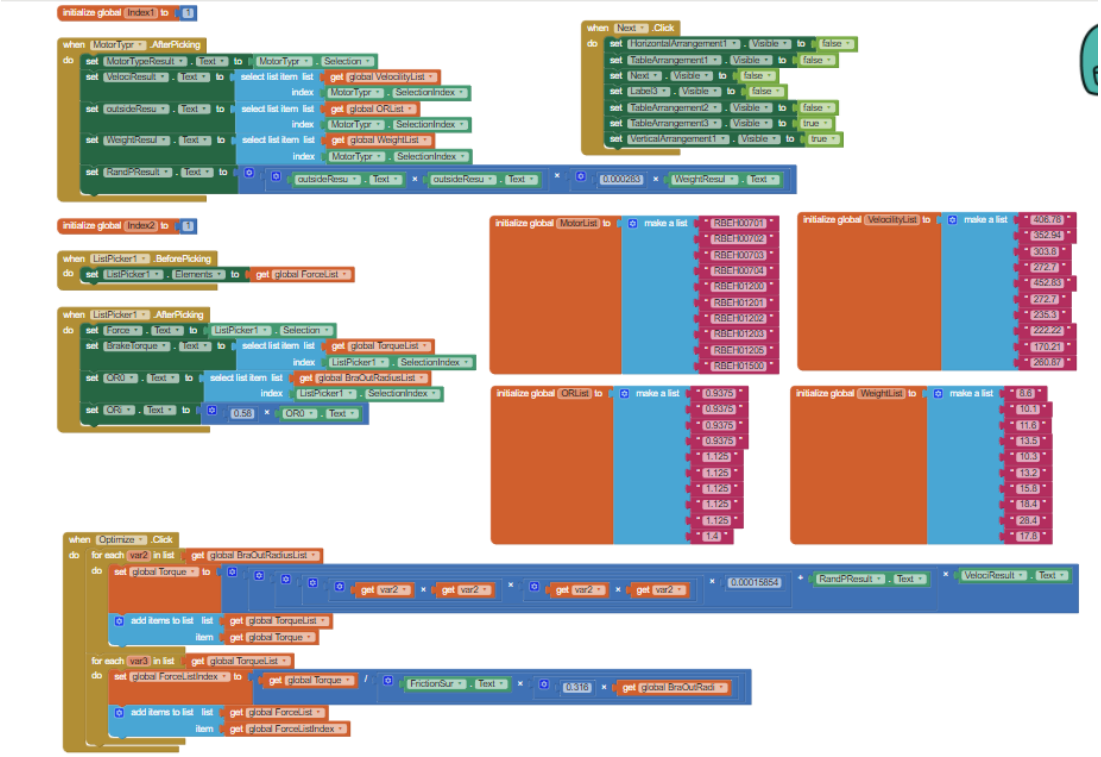


Figure 45. Disk Brake Optimization Blocks

5.5.6. Evaluation The app is fairly complex with all the input information and the data base of motor types and the complex calculations that must be carried out. It is a very good example of a successful app for a very complex engineering design problem.

5.6. Farm Crop Planting Problem

This is an example of a farm crop planting problem using Nonlinear Programming (NLP) on the NEOS server to solve the optimal mix of crops. It is an ambitious project but eminently suitable for AI2.

5.6.1. Introduction The app was created by Radha Dutta in 2015 who was a mathematics major in the Mie 379 course.

5.6.2. Problem This is a very important and challenging problem and of immense practical usefulness. The data were gathered from the UMass Student Farm program.

5.6.3. Mathematical Model Radha formulated a nonlinear programming problem. Some of the notation are:

$\rho_i :=$ Price that crop i sells for.
$\delta_i :=$ Total demand for crop i.
$\gamma_i :=$ crop yield/foot
$\mu_i :=$ Seeds or plants per foot.
$\sigma_i :=$ Number of seeds per packet.
$\omega_i :=$ Price of seed packet.
$\beta :=$ Total budget available.
$N :=$ Number of crops.

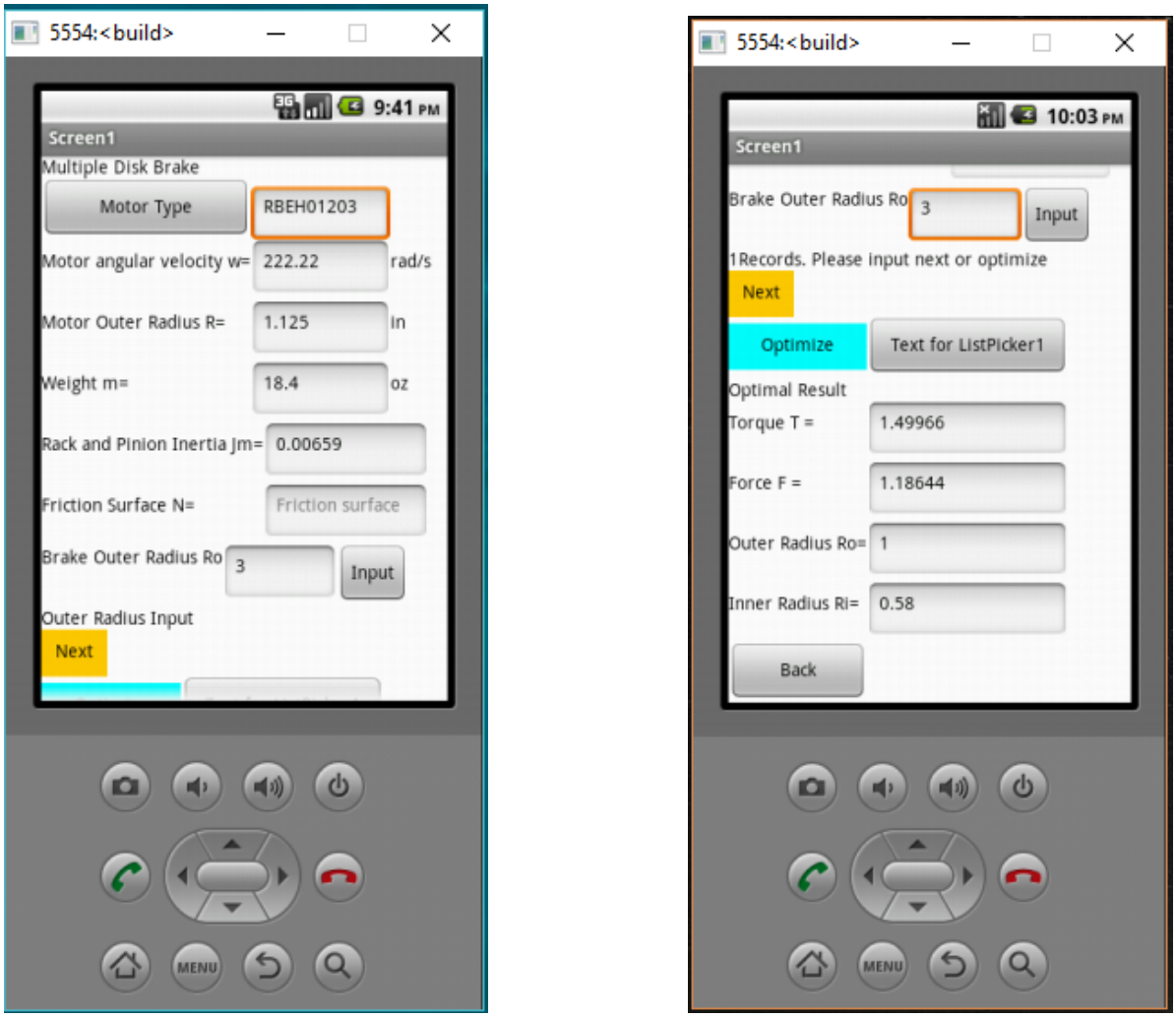


Figure 46. Disk Brake Optimization Blocks

$$\text{Maximize } Z = \sum_{i}^{N} \rho_i x_i \tag{75}$$

$$s.t. \qquad \sum_{i}^{N} \epsilon_i x) i \leq \beta_i where \epsilon_i = \left(\frac{2 x_i \mu_i}{\gamma_i 640 \sigma_i} \right) \tag{76}$$

$$\rho_i, \delta_i, \omega_i, \mu_i, x_i, \epsilon_i \geq 0 \tag{77}$$

5.6.4. Algorithm Part of the blocks programming is shown in Figure 47.

5.6.5. Demonstration There are five screens associated with the input process and all the parameters. There are a total of eight crops and associate cost and field size and seed requirements necessary to fill out the LP mathematical programming model. Three of the screens are shown in Figure 25.

Data for an example problem are given below:

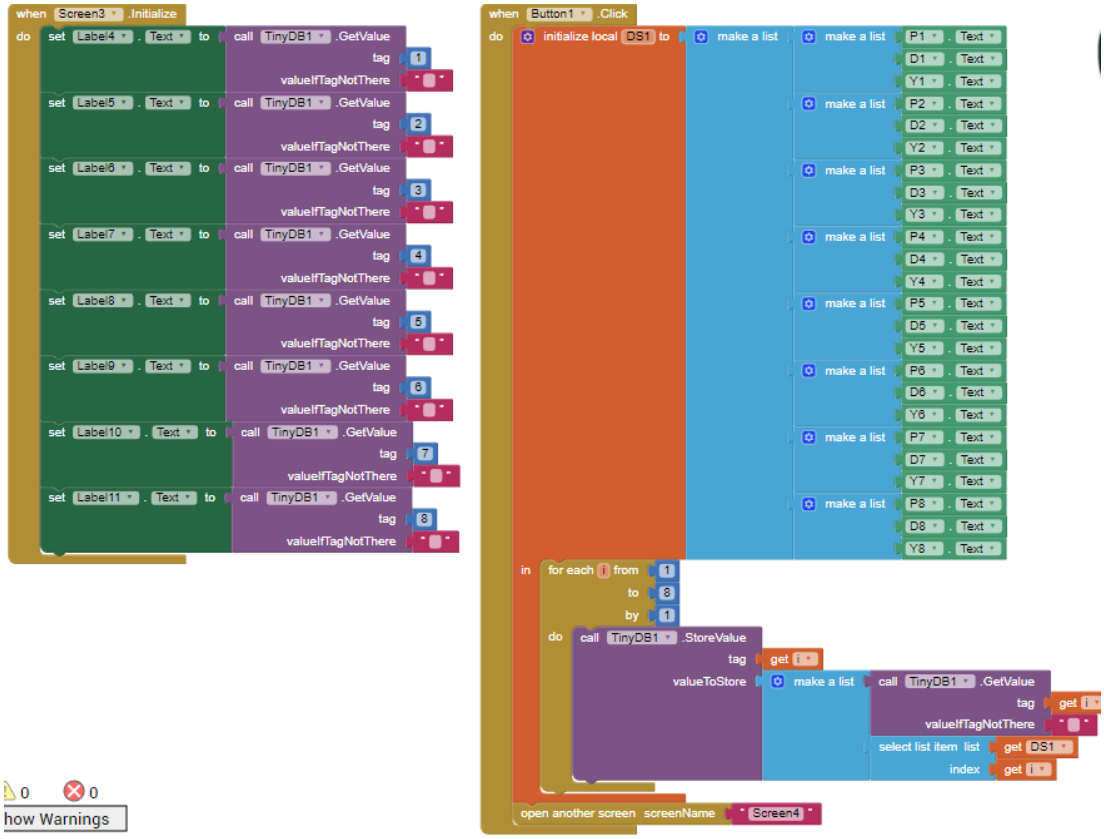


Figure 47. Farm Crop Blocks

$$
\begin{array}{l}
CropName \\
\\
Beets \\
Kale \\
Spinach \\
Radish \\
SweetPotato \\
Carrots \\
BokChoy \\
Cauliflower
\end{array}
\begin{bmatrix}
Price & Demand & Yield/\ foot & Seeds/\ foot & Seeds/\ Packet & Price/\ Packet & Epsilon \\
1 & 1690 & 1.25 & 6 & 1850 & 8.4 & 6.81081x10^{-5} \\
1.75 & 8600 & 1 & 1 & 3500 & 12.5 & 1.1607x10-5 \\
8 & 1050 & 0.4 & 12 & 2240 & 9.65 & 0.000403878 \\
2 & 875 & 0.50 & 12 & 10600 & 13 & 9.19811x10^{-5} \\
1.5 & 3995 & 3.5 & 1 & 1000 & 75 & 6.69643x10^{-5} \\
1 & 10183 & 1.5 & 16 & 100,000 & 101 & 3.36667x10^{-5} \\
2 & 720 & 0.75 & 2.5 & 5000 & 24.15 & 5.03125x10^{-5} \\
1.5 & 830 & 1 & 1 & 500 & 12.45 & 7.78125x10^{-5}
\end{bmatrix} \quad (78)
$$

The solution achieved from the NEOS server using the MINOS software was for the number of pounds for each crop:
Beets = 1690
Kale = 13360.50
Spinach = 1687.81
Radish = 1852.75
Sweet Potatoes =3995
Carrots =10183
Bok Choy]=3387.5
Cauliflower =1642.57

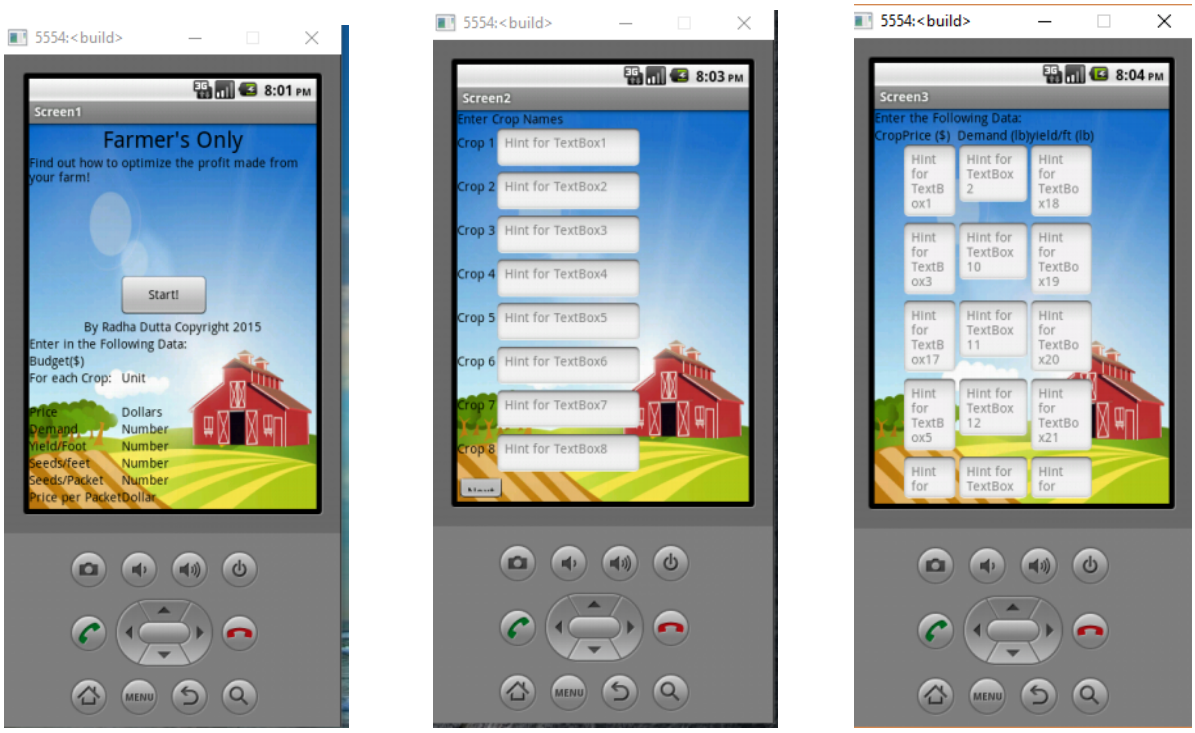


Figure 48. Farm Crop App Screens

Objective Value Z = 67692.30

It required 119 major iterations for the solution. Notice that some of the crops are equal to their demands, but others exceed their demands.

5.6.6. Evaluation Everything works very well for a complicated app. There is a long input process if one uses all eight crops. One can utilize fewer crops but one must put in zeros in all the cells where numbers are required otherwise the app will not execute on the NEOS server. This is true for most of the apps in that you must put in dummy answers in all the cells.

5.7. Erlang Loss M/G/c/c

This is a special type of queueing problem that has a great number of applications in telecommunications, industrial engineering, and operations research, and other related engineering and service system topics.

5.7.1. Introduction This queueing problem is one of the first queueing problems ever studied. It was originally started by A.O. Erlang for the Copenhagen Telephone Exchange in 1909 who was interested in determining the number of human operators to manage the telephone switch at the exchange. The formula he arrived at is the basis for the mathematical model of this app.

5.7.2. Problem For many systems, an arrival who finds all servers occupied is for all practical purposes lost to the system. We would like to minimize the probability of lost customers.

- Suppose we call an airline reservations system and all reservations clerks are busy.
- Suppose some one calls for an ambulance to respond to an accident or other emergency, and there are no ambulances available to respond?

Obviously, the ambulance problem is most serious. How can we insure that we have enough ambulances for an urban area to minimize this unfortuante circumstance.

5.7.3. Mathematical Model We have random arrivals (*i.e.* Poisson arrivals) and random service from a general probability distribution and a finite number of servers c and finite waiting room actually equal to the number of servers (c). This is called an $M/G/c/c$ queue for short.

The Erlang formula is:

$$P_c = \frac{(c\rho)^c/c!}{\sum_{i=0}^{c}(c\rho)^i/i!}, \rho = \lambda/c\mu$$

5.7.4. Algorithm A recursive procedure based upon a nonlinear equation is utilized to solve the problem where we take the input information and systematically find the threshold probability which satisfies the requirement for the system problem.

The blocks programming is shown in Figure 49.

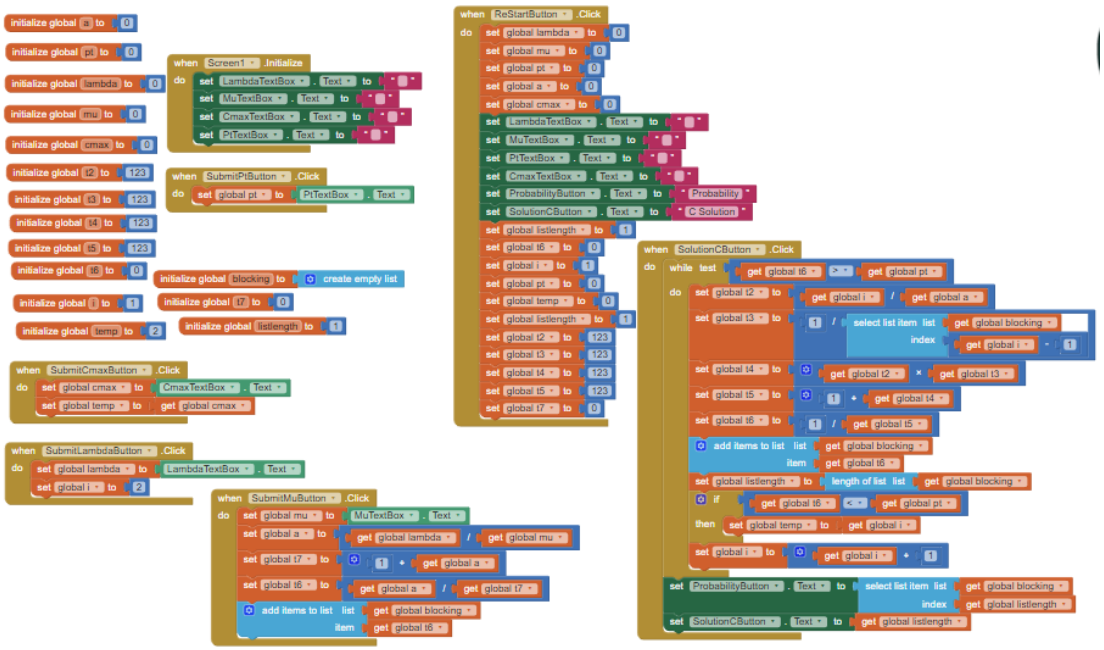


Figure 49. M/G/c/c App Programming

5.7.5. Demonstration Let's say that we are trying to plan the number of ambulances for an urban area. The idea for this example application came from Winston [5]. Suppose that on average 21 calls per hour are received at the 911 service center. An ambulance responding to a call takes on average 25 minutes to pick up a patient and deliver the patient back to the hospital. The ambulance is then able to respond to other emergency calls. How many ambulances should the hospital maintain in order to ensure that there is at most a 0.01 probability of not being able to respond immediately to an emergency call? The apps used for the input and the final solution are shown in Figure 50.

5.7.6. Evaluation The program works very well and is easy to use.

5.8. Warehouse Storage

Ways of determining the optimal area requirements of warehouse storage for a pallet rack system. Pallet racks are a very typical way of storing items in a warehouse. A very useful problem for determining warehouse storage in a factory layout context. We shall use calculus to solve this problem.

5.8.1. Introduction

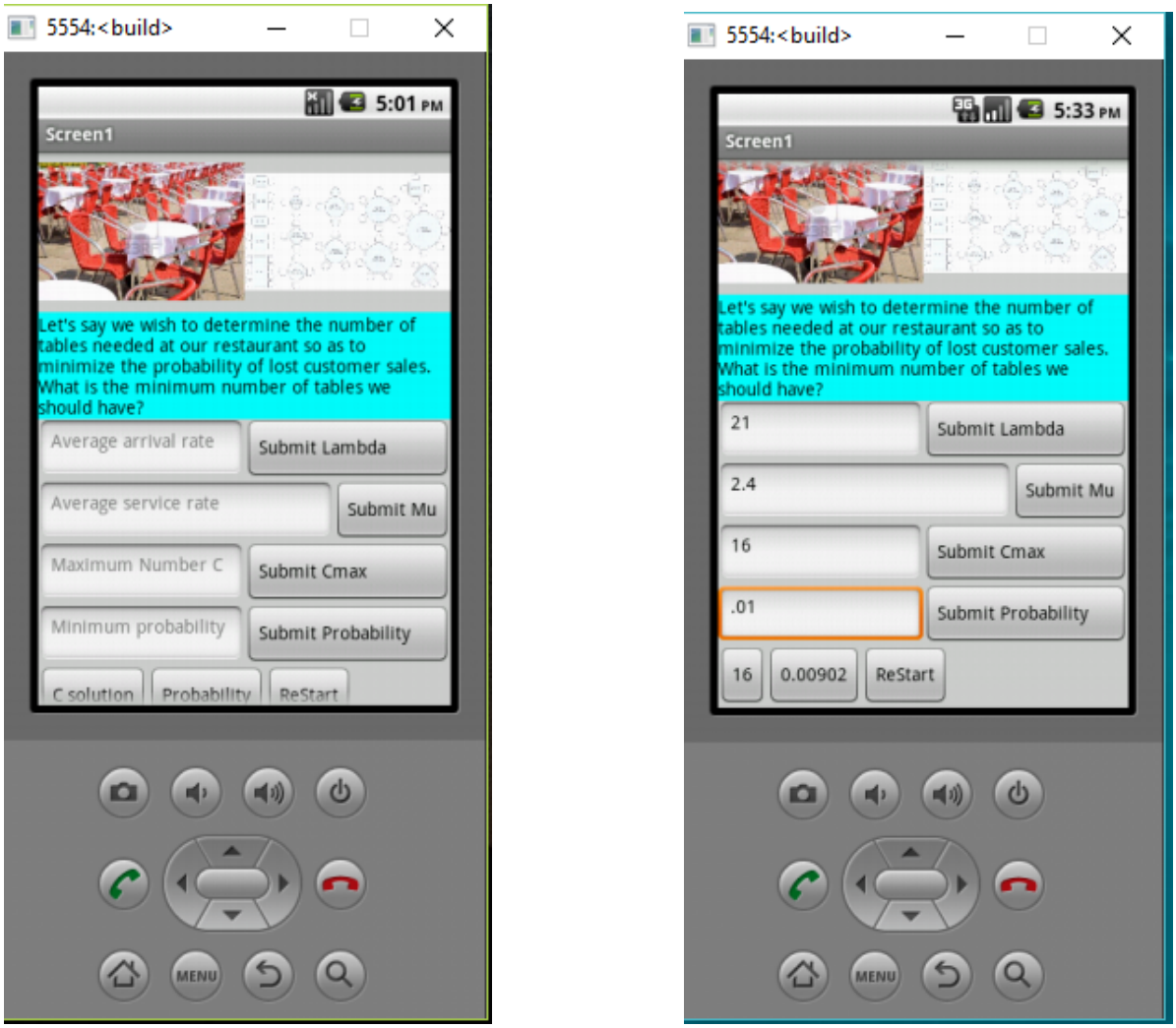


Figure 50. M/G/c/c App Screens

5.8.2. Problem

5.8.3. Mathematical Model

5.8.4. Algorithm

5.8.5. Demonstration

5.8.6. Evaluation

5.9. Newsvendor Problems

Determine the optimal quantity of a single perishable item subject to random demand over a single period with profit, setup, and salvage cost values. The calculus is also the mathematical model for understanding why this works.

5.9.1. Introduction

5.9.2. Problem

5.9.3. Mathematical Model

5.9.4. Algorithm

5.9.5. Demonstration

5.9.6. Evaluation

5.10. Jackson Network

Another queueing problem where one can determine the optimal service rates in a Jackson type queueing network. We can use calculus to solve this problem.

5.10.1. Introduction

5.10.2. Problem

5.10.3. Mathematical Model

5.10.4. Algorithm

5.10.5. Demonstration

5.10.6. Evaluation

6. AMPL Programming

Since many of the apps we need to solve require extensive computing resources over and above what App Inventor can supply, we need to elicit the support of external computing optimization algorithms. These AI2 apps will be appropriate for larger scaled problems.

6.1. Linear Programming

This is a very nice application app for more standard LPs.

6.1.1. Introduction

6.1.2. Problem

6.1.3. Mathematical Model

6.1.4. Algorithm

6.1.5. Demonstration

6.1.6. Evaluation

6.2. Assignment Problem

6.2.1. Introduction

6.2.2. Problem

6.2.3. Mathematical Model

6.2.4. Algorithm

6.2.5. Demonstration

6.2.6. Evaluation

6.3. Transportation Problem

6.3.1. Introduction

6.3.2. Problem

6.3.3. Mathematical Model

6.3.4. Algorithm

6.3.5. Demonstration

6.3.6. Evaluation

6.4. Travelling Salesman Problem

6.4.1. Introduction

6.4.2. Problem

6.4.3. Mathematical Model

6.4.4. Algorithm

6.4.5. Demonstration

6.4.6. Evaluation

7. Appendix: AI2 Introduction

We will give a short overview of AI2 as it relates to the optimization apps. There are many items of information on the web site of AI2 too numerous to mention.

1. Create a Google gmail account, if you do not already have one. Google will assist in your log on to App Inventor (AI) if you have a gmail account.
2. You may use either a PC or a Mac to run App Inventor.
3. Chapter 2 in Kamarani and Roy: *App Inventor 2 Essentials* shows in detail you how to set up AI2.
4. Go the the MIT App Inventor Program web site and read the instructions for how to start app inventor. `http://www.appinventor.mit.edu`

o You need to download a file that accesses App Inventor (AI).

o You should make sure you also have the **latest** java code (version 7+) to get classical AI to work smoothly.

o **David Wolber's book** `www.appinventor.org/book2` is very useful.

5. There is a tutorial site for the book which helps create the apps from the book. `http://www.appinventor.org/`
6. There are many other tutorial sites with more examples, called puravida apps: `http://puravidaapps.com/learn.php` There are many tutorials on this site.
7. There is a dynamic video for some very well-done and interesting examples. Highly recommended. Krishnendu Roy's website (*App Inventor 2 + Classic examples*): `http://coweb.cc.gatech.edu/ice-gt/1646`

7.1. Optimization Apps Problem Structure

Certain problems lend themselves to the optimization apps with AI2. These tend to be well-structured problems which need to be solved over and over again and would be helpful through a telephone communication system. The data requirements are not too extensive and it is something that one person can control. Of course as technology changes, these limitations may evolve, yet it seems that to be a one-person operation.

7.2. Tutorials

There are many tutorials and books available for learning AI2. Ones we have used in our course will be mentioned below. The web page for the course has hyper links to the tutorials.

7.3. Downloading Apps from the Website

The apps developed in the book are available from the course website.

References

[1] Cohon, J., 1978. **Multiobjective Programming and Planning.** Academic Press: New York.

[2] Lawler, E., 1976. **Combinatorial Optimization: Networks and Matroids.** Holt, Rinehart, and Winston.

[3] Rittel, H. and M. Webber, 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences,* **4**, 155-167.

[4] Russel, D., 1970. **Optimization Theiry.** W.A. Benjamin, Inc.: New York.

[5] Winston, W., 2004 **Introduction to Probability Models.** Fourth Edition. Thomson Brooks/Cole: Australia.