CS375 Final Project: Optimal Development Problem

By Nicholas DeNobrega and Samuel Buckler

Project Overview

- Selected Project:
  - Combine existing algorithms with implementation for solving a practical problem- Optimal Development Problem
- Algorithms being implemented:
  - Convex Hull: Given a set of points S, the convex hull of S is the smallest-area polygon which encloses S
  - Floyd's Algorithm: Given a graph G with vertices V and edges E, find the shortest path between every point to every other path. Known as the All-Paths Shortest Path Algorithm

What is a Convex Hull??

The convex hull of a set of points S in a plane is the smallest convex polygon that completely encloses all the points in S. Formally, it is the minimal convex set that contains S. The convex hull is often visualized as the shape formed when a rubber band is stretched around the outermost points of S and then released to snap tightly around them. For a polygon P, the convex hull is the smallest convex polygon that fully encloses P.

- If P is already convex, its convex hull is P itself, as no smaller enclosing convex shape can be formed.
- If P is non-convex (contains indentations), its convex hull will "smooth out" the indentations, including the points that create the indentations while forming a convex boundary

Figure 3

The Optimal Development Problem

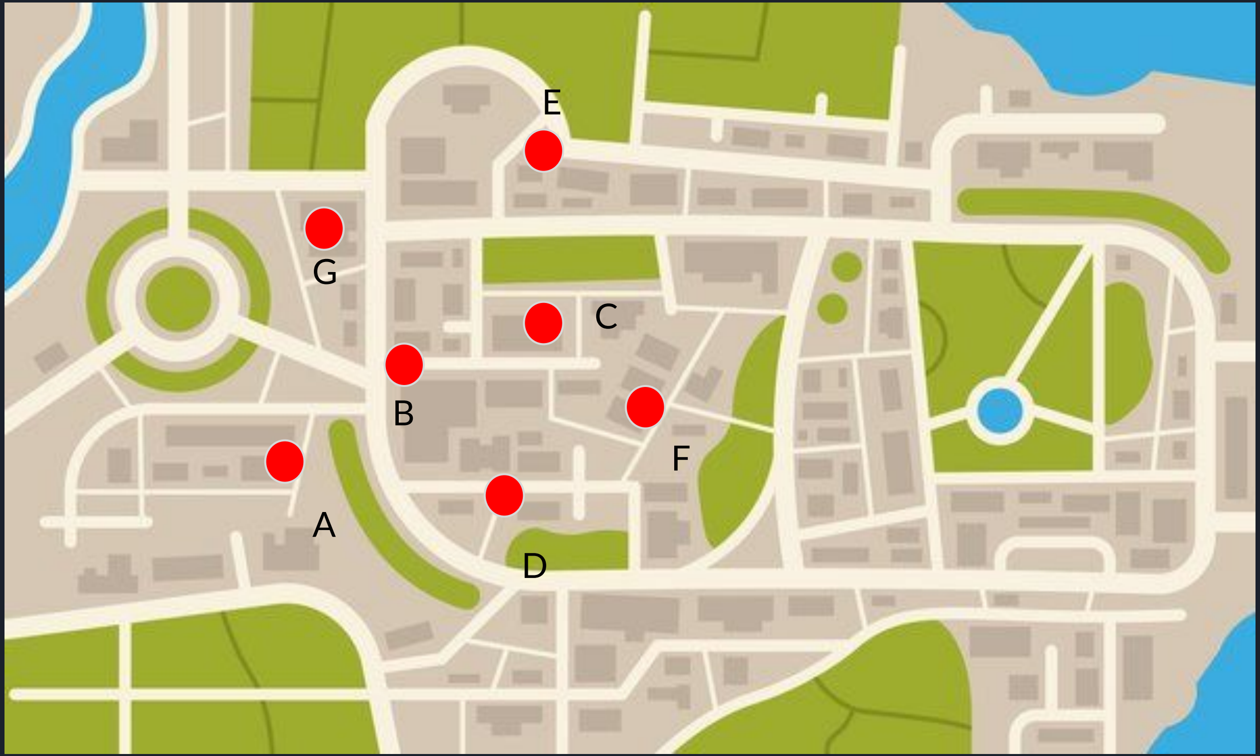
A builder is planning a residential development in a designated area and aims to include a set of points of interest (POIs)—such as parks, schools, and community centers—within the development. From this set of POI, the builder chooses a subset of the POI’s that he wants to include in the development. Beyond purchasing the land for the POIs themselves, the builder must also acquire and develop the land between the POIs to create a cohesive property. This land will accommodate future infrastructure such as roads, utilities, and community spaces.

The builder has a fixed budget and wants to maximize the number of POIs included in the development. At the same time, he must ensure the POIs are fully connected by an efficient roadway network. The total cost, including the cost of buying the land for the development, and the cost of building the roads, must not exceed his budget.

Optimal Development Problem Breakdown

- Land Development:
  - The purchased land must encompass as many POIs as possible and the space between them. This can be done by constructing a convex hull of the chosen POI
  - The cost of buying and developing the land (excluding the roadway system) is a function of the area encapsulated by the convex hull.
- Roadway Construction:
  - The roadways connecting the POIs should minimize total distance using a shortest-path network
  - The cost of the roadways is proportional to the total length of the connections x fixed cost of building a road
- Constraints:
  - The combined cost of land and road development must not exceed the budget
  - The roadway system should make the best use of the budget while maintaining connectivity between all selected POIs

Example Problem: Budget = 20k, Landcost = 2k/unit^2, Selected Points = A,B,C,D,F,G



Code Breakdown 1: Allowing Builder to select certain POI's

```
Available POI's:
1: POI Name: A, POI Latitude: 0, POI Longitude: 0
2: POI Name: B, POI Latitude: 1, POI Longitude: 1
3: POI Name: C, POI Latitude: 2, POI Longitude: 2
4: POI Name: D, POI Latitude: 2, POI Longitude: 0
5: POI Name: E, POI Latitude: 2, POI Longitude: 4
6: POI Name: F, POI Latitude: 3, POI Longitude: 1
7: POI Name: G, POI Latitude: 0, POI Longitude: 3

Enter the number of the POI you want to select (enter 0 to stop):
```

```cpp
// Given a set of POI's, allows "builder" to pick their budget and what points
// they want to buy/develop
void promptBuilder(const vector<Point> &allPoints, vector<Point> &selectedPoints, double &budget, double landPricePerUnit) {
    // Tracks what points have already been selected
    set<int> selectedIndices;

    while (true) {
        // Break if all points have been selected
        if (selectedPoints.size() == allPoints.size()) {
            cout << "No more POI's available to select.\n";
            break;
        }
        // Display points that have not been selected
        cout << "Available POI's:\n";
        for (int i = 0; i < allPoints.size(); ++i) {
            if (selectedIndices.find(i) == selectedIndices.end()) {
                cout << i + 1 << ": " << allPoints[i] << "\n";
            }
        }
        cout << "\nEnter the number of the POI you want to select (enter 0 to "
                "stop):\n";
        int choice;
        cin >> choice;
        if (choice == 0) break;
        // Check if the choice is valid
        if (choice < 1 || choice > allPoints.size() ||
            selectedIndices.find(choice - 1) != selectedIndices.end()) {
            cout << "Invalid choice. Try again.\n";
            continue;
        }

        selectedPoints.push_back(allPoints[choice - 1]);
        selectedIndices.insert(choice - 1);
    }

    cout << "\nPlease enter your budget for development: ";
    cin >> budget;
    while (budget <= 0) {
        cout << "Invalid budget. Please enter a positive number: ";
        cin >> budget;
    }

    cout << "\nYou selected the following POI's:\n";
    if (selectedPoints.empty()) {
        cout << "No POI's were selected.\n";
    } else {
        for (const Point &point : selectedPoints) {
            cout << point << "\n";
        }
    }
    cout << "Your Selected budget is $" << budget << endl;
    cout << "The price of land per unit is $" << landPricePerUnit << endl;
```

Code Breakdown 2: findOptimalDevelopmentPlan()

- Iterate through all possible subsets of POI’s
- Construct convex hull of current subset by calling getHullDNC
- Calculate area of current convex hull using Shoelace Formula, also known as Gauss’s Area Formula, and multiply by the cost of the land

$$\text{Area} = \frac{1}{2}\left|\sum_{i=1}^{n}(x_i y_{i+1} - y_i x_{i+1})\right|$$

- If the cost of purchasing the land is > budget, move on to next subset
- If not, create a graph of the current subset and call calculateRoadCost, which is a modified Dijkstra's algorithm to calculate cost of connecting the POI
- If the cost of purchasing the land + cost of building roadways <= budget, update the best found hull and best cost to current hull and cost

```cpp
// Given a chosen set of POI by the builder, the builder's budget, and a land
// price, gets the most optimal set of POI that a builder can buy. Optimal set =
// maxamize number of POI without exceeding budget, where the price includes
// cost of buying the land, and building roads to connect the POI
void findOptimalDevelopmentPlan(vector<Point> &points, int budget, double landPricePerUnit) {
    vector<Point> bestHull;
    double bestLandCost = 0;
    double bestRoadCost = 0;
    double bestCost = 0;
    int n = points.size();

    //Iterate over all subsets of points
    for (int mask = 1; mask < (1 << n); ++mask) {
        // 1. Create current subset of points
        vector<Point> currentSubset;
        for (int i = 0; i < n; ++i) {
            if (mask & (1 << i)) currentSubset.push_back(points[i]);
        }
        // 2. Get Convex Hull of current subset
        vector<Point> currentHull = getHullDNC(currentSubset);
        // 3. Calculate area of current Convex Hull
        int currentLandArea = calculateConvexHullArea(currentHull);
        // 4. Calculate land cost of area of current Convex Hull
        int currentLandCost = currentLandArea * landPricePerUnit;
        // 5. If cost of current Convex Hull exceeds budget, do not update
        // bestHull with currentHull, else continue to calculating cost of roads
        if (currentLandCost > budget) continue;
        // 6. Create a graph to represent the current subset of points
        vector<vector<int>> graph = generateGraph(currentSubset);
        // 7. Calculate cost of building roads to connects all points
        double currentRoadCost = calculateRoadCost(graph);
        // 8. Calculate cost of current Convex Hull (Cost of land) + current road
        //    cost (Cost of building roads connecting POI)
        double currentCost = currentLandCost + currentRoadCost;
        // 9. If currentCost is <= budget, we know we can afford to buy the land for
        //    the current subset of POI and build roads connecting all the POI, so we
        //    update our bestHull to currentHull, and our bestCost to currentCost, then
        //    move on to next subset of POI
        if (currentCost <= budget && currentCost > bestCost) {
            bestHull = currentHull;
            bestLandCost = currentLandCost;
            bestRoadCost = currentRoadCost;
            bestCost = currentCost;
        }
    }

    // 10. Print the optimal solution
    cout << "POI in Optimal Solution:\n";
    for (const Point &p : bestHull) {
        cout << p << endl;
    }
    cout << "\nLand Cost: " << bestLandCost;
    cout << "\nRoadway Cost: " << bestRoadCost;
    cout << "\nTotal Cost: " << bestCost << endl;
}
```

Code Breakdown 3: getHullDNC()

- Takes in set of points
- If # of points is < 6, we call getHullBrute(), which is the brute force method of finding the convex hull
- If > 6, recursively find convex hulls of left and right halves, then merge the two convex hulls together by calling mergeHulls()
- Time complexity ~ O(nlogn),

```cpp
// Given set of points, construct a Convex Hull of those points
vector<Point> getHullDNC(vector<Point> points) {
    //Base Case: If # points is < 6, use Brute Force algorithm to get Convex Hull
    if (points.size() <= 5){
        return getHullBrute(points);
    }

    vector<Point> left, right;
    //Add left half of points to "left"
    for (int i = 0; i < points.size() / 2; i++){
        left.push_back(points[i]);
    }

    //Add right half of points to "right"
    for (int i = points.size() / 2; i < points.size(); i++){
        right.push_back(points[i]);
    }

    // Recursively get Convex Hulls for left and right sets
    vector<Point> leftHull = getHullDNC(left);
    vector<Point> rightHull = getHullDNC(right);
    // Merge the two Convex Hulls
    return mergeHulls(leftHull, rightHull);
}
```

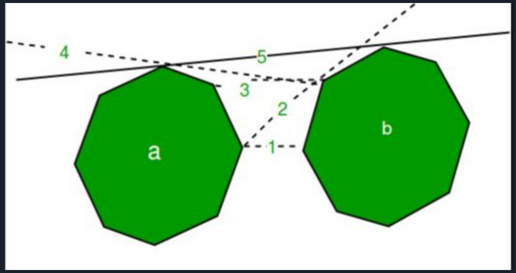
Code Breakdown 4: getHullBrute()

- Iterates through all possible pairs of points
- For each pair, we construct a line that connects these points using formula ax + by = c
- If all of the remaining points are either on one side of the line or the other (but not both sides), then this line is a boundary of the hull.
- Finally, add these points to ret and sort in counterclockwise order
  - Calculate midpoint as mid, which is average of all boundary points.
  - Given points P and Q, with midpoint M, find the cross-product of PM X QM where PM and QM are vectors.
  - Positive result implies q is counterclockwise to p, so therefore q comes before p.
  - Negative result implies p is counterclockwise to q, so therefore p comes before q.
- Time Complexity = O (n^3), loop through all pairs, for each pair loop through every point

```cpp
// Brute force algorithm to find Convex Hull for < 6 points
vector<Point> getHullBrute(vector<Point> points) {
    //Set of boundary points of the Convex Hull
    set<Point> boundaryPoints;

    //Loop through all set of points
    for (int i = 0; i < points.size(); i++) {
        for (int j = i + 1; j < points.size(); j++) {
            int x1 = points[i].x, x2 = points[j].x;
            int y1 = points[i].y, y2 = points[j].y;

            /*
            Line ax + by = c between 2 points (x1, y1) and (x2, y2)
            where a = y2-y1, b=x1-x2, c=x1y2-y1x2
            All points on one side of the line: ax + by > c
            All points on the other side: ax + by < c
            */
            int a1 = y2 - y1;
            int b1 = x1 - x2;
            int c = x1 * y2 - y1 * x2;
            int pointsAboveLine = 0, pointsBelowLine = 0;

            //See what side of the line any of the points are on
            for (int k = 0; k < points.size(); k++) {
                if (a1 * points[k].x + b1 * points[k].y >= c) pointsBelowLine++;
                if (a1 * points[k].x + b1 * points[k].y <= c) pointsAboveLine++;
            }

            //If all the points are either on one side of the line or the other,
            //Then the two points being examined are part of the hull's boundary
            if (pointsBelowLine == points.size() || pointsAboveLine == points.size()) {
                boundaryPoints.insert(points[i]);
                boundaryPoints.insert(points[j]);
            }
        }
    }
    vector<Point> ret;
    for (Point p : boundaryPoints){
        ret.push_back(p);
    }

    // Calculate the midpoint
    mid = {0, 0};
    for (const Point& p : ret) {
        mid.x += p.x;
        mid.y += p.y;
    }
    mid.x /= ret.size();
    mid.y /= ret.size();
    // Sort in counterclockwise order
    sort(ret.begin(), ret.end(), [&](const Point& p1, const Point& q1) {
        Point p = {p1.x - mid.x, p1.y - mid.y};
        Point q = {q1.x - mid.x, q1.y - mid.y};
        return (p.y * q.x < q.y * p.x);
    });
    return ret;
}
```

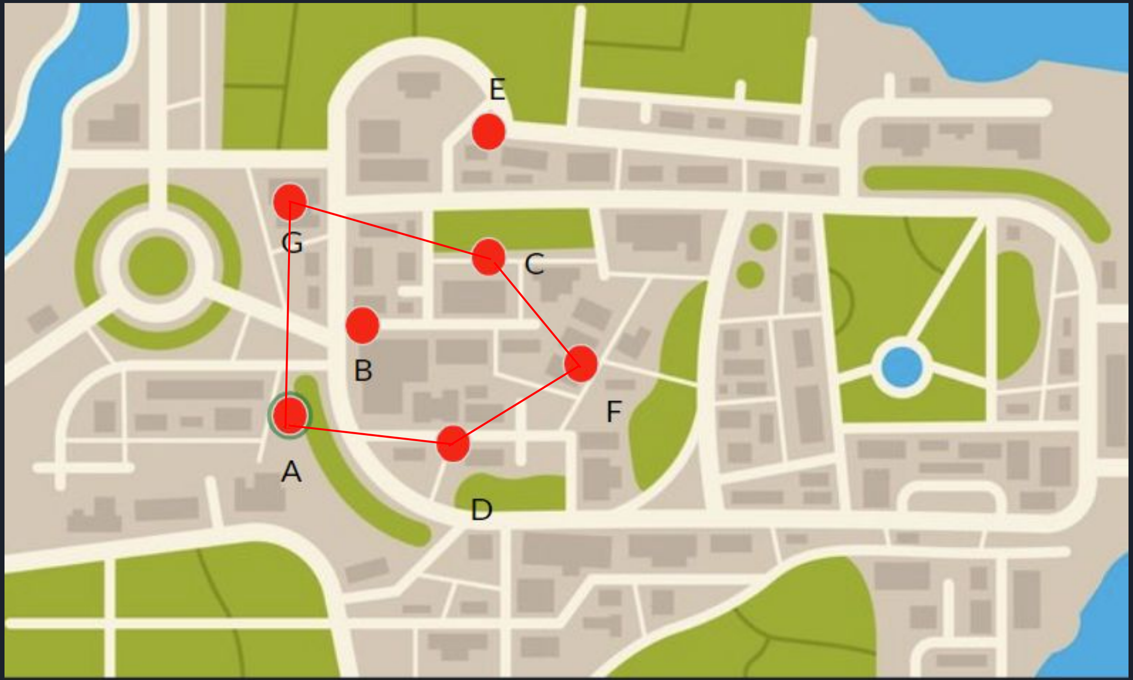
Code Breakdown 5: mergeHulls()

- First, finds the rightmost point of left hull and leftmost point of right hull and store as iL and iR
- We iteratively adjust left and right points to find upper tangent of the two hulls. I.E the line connecting theme that does not cross either hull
- For each iteration to calculate upper tangent:
    - Call orientation() to determine if the line connecting the current left and right points goes through interior of hulls
    - If so, the left point gets adjusted one turn counterclockwise, or the right point gets adjusted one turn clockwise
- Same logic is applied for finding lower tangent except we shift left point clockwise and right point counterclockwise
- Finally, to merge we traverse left hull from upper tangent to lower tangent and add to ret, then traverse right hull from lower tangent to upper tangent and add to ret
- Time complexity = O(n)




```cpp
vector<Point> mergeHulls(vector<Point> leftHull, vector<Point> rightHull) {
    int n1 = leftHull.size();
    int n2 = rightHull.size();
    int iL = 0;
    int iR = 0;

    //Find rightmost point of leftHull, store in iL
    for (int i = 1; i < n1; i++)
        if (leftHull[i].x > leftHull[iL].x) iL = i;
    // Find leftmost point of rightHull, store in iR
    for (int i = 1; i < n2; i++)
        if (rightHull[i].x < rightHull[iR].x) iR = i;


    int indL = iL;
    int indR = iR;
    bool done = false;
    /*
    Given two starting points a (indL) and b (indr), where a is the rightmost left hull point and b is the leftmost right hull point,
    The goal is to find points a and b such that AB does not cross through the left hull or the right hull
    This would represent the upper tangent line
    */
    while (!done) {
        done = true;
        //This while loop determines the point A (indL) that ensures AB does nt cross through the left hull
        //Moding by n1 (# points in left hull) ensures that the point incrementation does not go out of bounds of left hull
        while (orientation(rightHull[indR], leftHull[indL], leftHull[(indL + 1) % n1]) >= 0){
            indL = (indL + 1) % n1; //counter clockwise turn
        }
        //This while loop determines the point B (indR) that ensures AB does not cross through the right hull
        //Modding by n2 (# points in right hull) ensures that the point incrementation does not go out of bounds of right hull
        while (orientation(leftHull[indL], rightHull[indR], rightHull[(n2 + indR - 1) % n2]) <= 0) {
            indR = (n2 + indR - 1) % n2; //clockwise turn
            done = false;
        }
    }
    // Find the lower tangent, logic is the same as upper tangent calculations except it does it in reverse order
    int upperL = indL;
    int upperR = indR;
    indL = iL;
    indR = iR;
    done = false;
    while (!done){
        done = true;
        while (orientation(leftHull[indL], rightHull[indR], rightHull[(indR + 1) % n2]) >= 0){
            indR = (indR + 1) % n2; //counter clockwise turn
        }
        while (orientation(rightHull[indR], leftHull[indL], leftHull[(n1 + indL - 1) % n1]) <= 0) {
            indL = (n1 + indL - 1) % n1; //clockwise turn
            done = false;
        }
    }
    //Store the merged Convex Hulls in ret in counter-clockwise order
    int lowerL = indL;
    int lowerR = indR;
    vector<Point> ret;
     //Traverse leftHull from upper tangent to lower tangent and add these points to ret
     int ind = upperL;
     ret.push_back(leftHull[upperL]);
     while (ind != lowerL) {
         ind = (ind + 1) % n1;
         ret.push_back(leftHull[ind]);
     }
     //Traverse rightHull from lower tangent to upper tangent and add these points to ret
     ind = lowerR;
     ret.push_back(rightHull[lowerR]);
     while (ind != upperR) {
         ind = (ind + 1) % n2;
         ret.push_back(rightHull[ind]);
     }
     return ret;
}
```

Example Problem: After Finding Best Convex Hull, Budget = 20k, Landcost = 2k/unit^2, Selected Points = A,B,C,D,F,G



Results/Conclusion

- Results shown when running code with given example (Budget of 20k, a land price of 2k, and a chosen set of points A,B,C,D,F,G)
- To be implemented:
  - The roadway system that connects points with each other in the most efficient way possible
  - This will be done with a modified Floyd’s Algorithm to calculate the cost of building the roadways

```
You selected the following POI's:
POI Name: A, POI Latitude: 0, POI Longitude: 0
POI Name: B, POI Latitude: 1, POI Longitude: 1
POI Name: C, POI Latitude: 2, POI Longitude: 2
POI Name: D, POI Latitude: 2, POI Longitude: 0
POI Name: F, POI Latitude: 3, POI Longitude: 1
POI Name: G, POI Latitude: 0, POI Longitude: 3
Your Selected budget is $15000
The price of land per unit is $2000
POI in Optimal Solution:
POI Name: F, POI Latitude: 3, POI Longitude: 1
POI Name: C, POI Latitude: 2, POI Longitude: 2
POI Name: G, POI Latitude: 0, POI Longitude: 3
POI Name: A, POI Latitude: 0, POI Longitude: 0
POI Name: D, POI Latitude: 2, POI Longitude: 0

Land Cost: 12000
Roadway Cost: 0
Total Cost: 12000
```