Competitive programmintg
Greedy algorithms

Prof. Rhadamés Carmona

Last revision: 26/02/2020

Content

- Concept
- Elements of a greedy algorithm
- Examples
  - Kruskal, Prim y Dijkstra
  - TSP
  - Coin change
  - Polygon Triangulation
  - Activities selection
  - Other examples
  - A*

Concept

- A voracious or greedy algorithm at each step selects a locally optimal alternative through a heuristic, with the hope of obtaining the optimal global solution.
- It's not exhaustive. For certain problems the greedy strategy results in the optimal solution, while for other problems it does not.

Concept

- They are efficient algorithms; select at each step the best candidate to belong to the solution.
- As the algorithm progresses, there are two sets of elements: those that are in the solution, and those that are not in the solution.
- Typically there is a function that you want to maximize or minimize.

Concepto

```
solution greedy(candidates &c) {
        solution s();
        while (!c.isEmpty() && !is_solution(s,c)) {
                item = best_item(c);
                c -= item;
                if (feasible(s, item))
                        s += item;
        }
        return (is_solution(s,c) ? s : solution::empty();
}
```

Elements of a greedy algorithm

1) Candidates: We must have a set of candidates
2) Objective function: map a solution to a cost or value
3) Function that indicates whether the assembly is a complete solution (is_solution)
4) Selection function (best_Item) that indicates the best alternative
5) Function that checks whether an item can be added to the partial solution to approach the solution (feasible)

Kruskal

- Algoritmo de Kruskal: usado para encontrar el árbol de expansión mínimo en un grafo.
- Se ordenan todos los arcos del grafo según su weight.
- Partiendo de un bosque de árboles de un nodo (los vértices), se agregan un arco a la vez si no forma ciclo, hasta completar V arcos (V-1 vértices).
- Cada arco agregado “une” dos árboles del conjunto.

Ejemplo: Kruskal

```
queue<edge*> &mst KruskalMST(graph &g, PriorityQueue<edge *> &sortedEdges)  {
        disjointSet <int>ds;            // conjuntos disjuntos
        queue<edge*>  mst;              // solucion
        while (!pq.empty() && mst.size() < g.size()-1)    {          // O(mlog.m)
                edge *e = pq.deleteMin();                            // best & delete
                if (ds.root(e->left) != ds.root(e->right))  {        // feasible?
                        ds.unionFind(e->left, e->right, true);       // Merge
                        mst.push_back(e);                            // s += item;
                }
         }
        return mst.size() == g.size() ? mst : queue<edge*> ();
}
solution greedy(candidates &c) {
        solution s();
        while (!c.isEmpty() && !is_solution(s,c)) {
                item = best_item(c);  c -= item;
                if (feasible(s, item)) s += item;
        }
        return (is_solucion(s,c) ? s : solution::empty();
}
```

Dijkstra

- Used to get the minimum paths in a graph, starting from a source node "s".
- At each step, add the closest vertex to "s" based on distance.

TSP

- Given n cities, how to visit the n cities visiting each city only once, with the lowest cost (in distance or in money).
- It's an NP-complete problem.
- The greedy solution (no optimal) is to start from city A, and take the nearest unvisited adjacent B city. Then do the same for B ... Until all cities are visited (hopefully...).

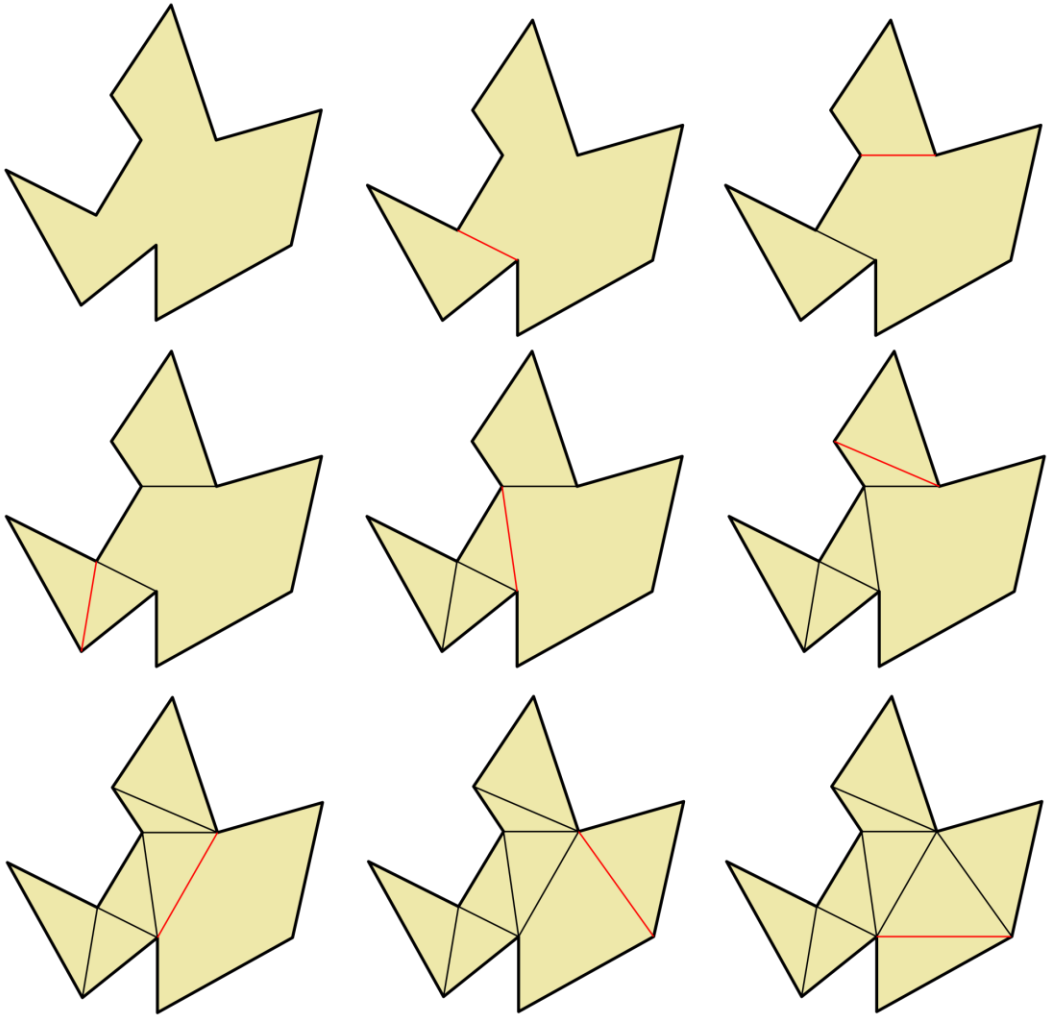
Coin change

- Given a monetary cone, give the coin change with as few coins as possible.
- The greedy strategy at each step seeks to deliver the highest denomination coin or currency that "fits" within the return. For example, if the values are 100, 50, 20,10, 5, 1, then try v/100, then (v%100)/50...
- Optimal for common monetary cones. Does not work on rare monetary cones (7, 3, 2 or 9,6,5,1 with a change of 11)

Polygon Triangulation

- Greedy polygon triangulation
- Candidate set: Every possible vertex pair within the polygon.
- Best candidate: pair of closest vertices (shorter edge).
- Feasibility: The new edge cannot cut to another edge previously added to the result.
- Solution: When there are no more arcs to check.
- The solution is not necessarily optimal in terms of desirable properties of a triangulation (e.g. Delaunay triangulation).

Polygon Triangulation



Activity selection

- Given a set of activities indicating the start and completion time of each (Si, Ei), select the maximum number of activities that a person can perform.
- Two activities i, j are “compatible” if $Sj \geq Ei$ or $Si \leq Ej$.
- They are incompatible otherwise.
- The problem is finding the largest possible compatible set.

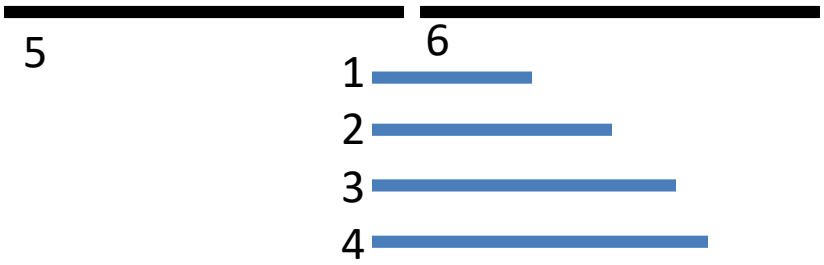
Activity selection

- The brute force solution by backtracking is to consider the two possibilities: x is in the solution, or is not. As there are n elements, this is $O(2^n)$.
- The greedy "naive" solution is to sort the activities by duration (Ei-Si) or by start time and insert them into the partial solution, provided that it does not "overlap" with any activity in the partial solution. This is suboptimal; possibly lousy!

Activity selection

We sorted by start time, but the best solution is the three shortest
So we could think of sorting by duration

The best solution is the two that take longer... so ordering by duration doesn't work either



Optimal solution: Select the next activity that ends earlier, including those that begin after the selected previous activity is complete. We start by selecting the activity that ends first.

Activity selection

- The reason is that the activity that finishes first is the one that leaves the most time for the other possible activities. Hence the optimality.
- To make searches easier, we sort the items ascending by Ei (end time). Unfortunately this is O(nlogn). But the rest of the algorithm is O(n).

Activity selection

```
void activity_selection(int s[], int e[], int n)
{
        int k=0; sort_asc_by_end(s,e,n);
        printf("%d ", 0); // first activity is selected
        for(int i=1; i<n; i++)
                if(s[i] >= e[k])
                {
                        printf("%d ", i);
                        k=i;
                }
}
```

Other examples

- **Huffman code**: to compress. Each word is assigned a unique code. Shorter binary codes are for items with more repetitions. Suffixes between texts cannot be repeated to avoid confusion when decompressing. https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/
- **Fractional Knapsack**: the backpack problem, but it allows you to take a fraction of an object (and not necessarily complete objects). Objects are sorted by the ratio between value/weight.

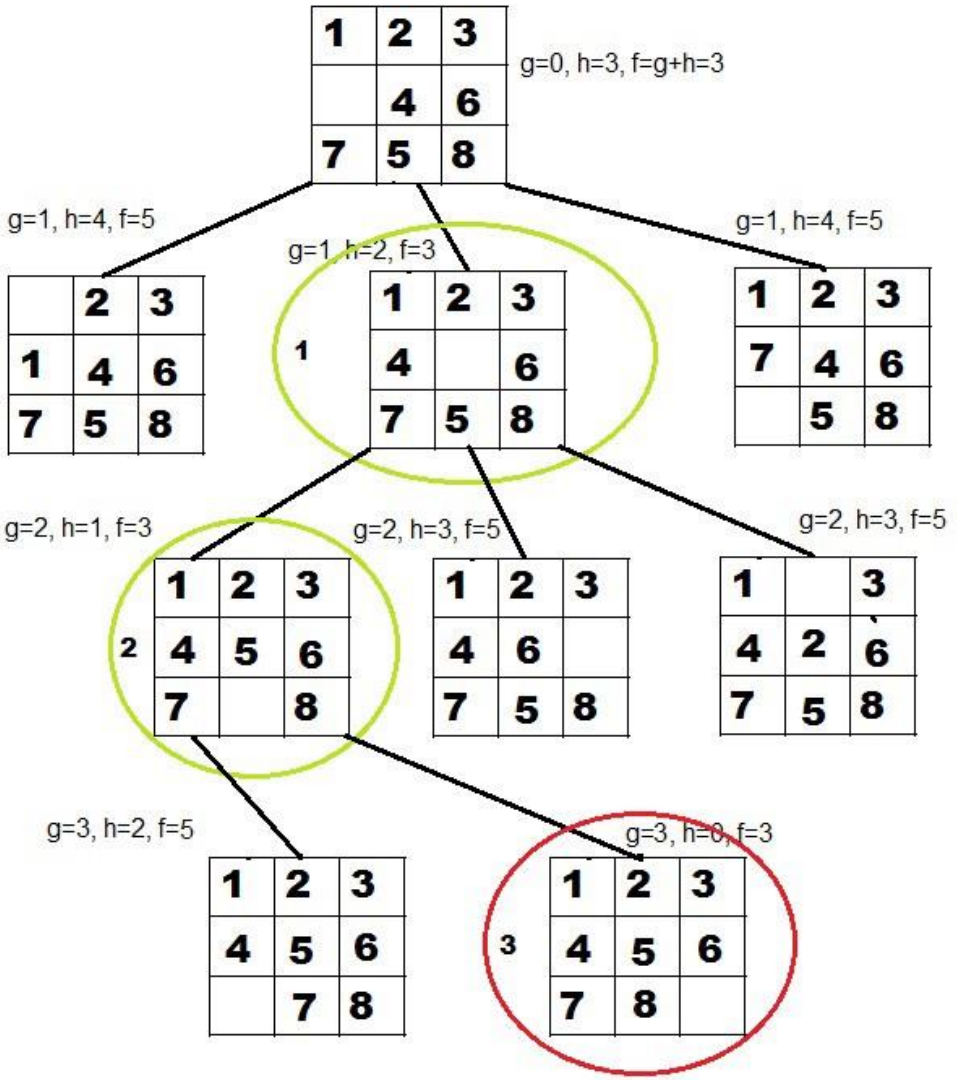
A*

- It is a voracious algorithm that works with heuristics to reduce search times.
- At each step, candidates are sorted according to the cumulative cost g(n) and a heuristic h(n); the latter to help you visit the most promising nodes first. The function of the priority queue is f(n)=g(n)+h(n).
- The first item in the queue is extracted, and its adjacent nodes are updated and added to the priority queue.

A*

- The algorithm continues until the priority queue is empty, or until it is on a node with a lower value of f(n) than any other node in the queue.
- Heuristics are usually related to "distance or cost" (under some metric) to the destination. Ideally h(n)=0 when you arrive at the destination. For example, how many pieces are not in place in an 8-puzzle? g(n) could be the accumulated path length to node or state n-th.

A*



In this example, h(n)=#pieces that are not in position. Another possible function h(n) is the sum of the distances of each part to its final position. Again h(n)=0 when you reach the solution.

Problems

https://www.spoj.com/problems/GERGOVIA/
https://www.spoj.com/problems/BLOPER/
https://www.spoj.com/problems/BLINNET/
https://www.spoj.com/problems/MSCHED/
https://www.spoj.com/problems/SNGINT/
https://www.spoj.com/problems/BUSYMAN/
https://www.spoj.com/problems/ACTIV/ (DP)