1 Pricing

We describe one application of reoptimization. We will see more applications in the sections on cut generation and column generation in the chapter on integer linear programming.

Every bounded LP has a basic feasible solution. In a basic feasible solution, only basic variables are non-zero and all non-basic variables are non-zero. It would be nice if we never had to consider the variables that are non-basic in the optimal solution. Pricing tries to achieve exactly this. It uses the following ideas.

- Identify variables that are likely to be non-basic in an optimal solution.
- Solve the LP without them (equivalently consider only bases containing none of the dropped variables).
- Check the reduced costs of the dropped variables. If all of them have non-positive reduced cost (this assumes that we are solving a maximization problem), declare the current solution optimal.
- If some of them have positive reduced cost, add them to the linear program and continue with phase II of the primal simplex algorithm.

Let us see a concrete example. We consider again the weighted perfect matching problem. Assume that we are given n red points R and n blue points B in the plane. The goal is to find a perfect matching between the red points and the blue points of minimum weight. The weight of the edge connecting points $r \in R$ and $b \in B$ is the Euclidean distance between r and b.

The naive way of solving the problem is set up the complete bipartite graph with n nodes on each side and to solve the weighted perfect matching problem on this graph. The approach has space requirement $\Omega(n^2)$ and hence we will not be able to go beyond a few thousand points. Assuming that we need 20 Bytes per edge, the space requirement for $n = 10^4$ amounts to $20 \cdot 10^8 = 210^9$ Bytes; 2 Gigabytes is a lot of memory.

An alternative approach using pricing is as follows. It is reasonable to assume that most nodes are matched with nearby points from the other color. We therefore fix a constant k, say $k = 50$, and determine for every point (red or blue) the k nearest neighbors. This can be done using data structures from computational geometry and then the time required is $O(n \log n + kn)$. Alternatively, we might use exhaustive search. This has time $O(n^2 k)$ or $O(n^2 \log k)$ depending on the implementation.

We now have a graph with $m \le 2kn$ edges. We compute a weighted perfect matching and (as a byproduct) node potentials. We then check for every pair (r, b) whether $dist(r, b) \ge y_r + y_b$. If the inequality is satisfied for all edges, the current matching is optimal (we are computing a minimum cost matching and hence want $dist(r, b) \ge y_r + y_b$ for all (r, b).

Checking Optimality: The easiest way to check whether $dist(r,b) \geq y_r + y_b$ for all (r,b) is by looking at all pairs (r,b). This takes time $O(n^2)$. We could extend the graph by all edges with $dist(r,b) < y_r + y_b$. The number of such edges is potentially large and hence we could again arrive at a dense graph. It is better to add only a small number of edges per node. Namely, those with large $y_r - y_b - dist(r,b)$.

There is an alternative method for checking optimality. If $dist(r,b) < y_r + y_b$ then $dist(r,b) < 2\max(y_r, y_b)$. We can therefore proceed as follows. For every red point r we generate the list of b with $dist(r,b) < 2y_r$ and for every blue point b we generate the list of r with $dist(r,b) < 2y_b$. In this way we obtain a list C of candidates. We then check all pairs (r,b) in the candidate list for $dist(r,b) < y_r + y_b$. The hope is that the candidate list contains only a small subset of the pairs.

The candidate list can be generated quickly by geometry data structures. Observe that for fixed r the task of finding all b with $dist(r,b) < 2y_r$ amounts to finding all blue points contained in the disk centered at r and having radius $2y_r$. This is called a *circular range query* and there are efficient solutions for it. For example, the LEDA data type *point_set* supports circular range queries.

1.1 An Implementation

I will next report about a crude implementation of pricing for computing the minimum cost perfect matching between n red points and n blue points in the plane. The main program is as follows: I generate n red and n blue points randomly in the square with side length 10000 and generate a bipartite graph with n red nodes (corresponding to the red points) and n blue nodes (corresponding to the blue points). No edges are generated yet. Then we compute a minimum cost perfect matching, once with and once without pricing. I report CPU time and wall time (elapsed time). I report the latter because the implementation without pricing uses up so much space that it has to use secondary memory even for moderate n (below 2000 points on my machine). The use of secondary memory slows down the program tremendously. .

⟨*pricing_test.c*⟩≡

```
#include <LEDA/geo_alg.h>
#include <LEDA/graph_alg.h>
#include <LEDA/map.h>
#include <LEDA/point_set.h>
#include <LEDA/node_map2.h>
#include "walltime.h"
#include <math.h>

double distance(const point& p, const point& q)
{ double D = 1000 * p.distance(q);
  D = (int) D;
  if ( D < 0 ) D = 0;
  return D;
```

```
}
⟨insert⟩

int main(){
  //int n = read_int("number of points = ");

for (int n = 100; true; n *= 2)
  { cout << "\n\nnumber of nodes = " << n;

float T = used_time();

list<point> B;
random_points_in_square(n,10000,B);

list<point> R;
random_points_in_square(n,10000,R);

GRAPH<point,double> G;
point b,r;
map<point,node> point_to_node;
list<node> BL, RL;

forall(b,B) BL.append(point_to_node[b] = G.new_node(b));
forall(r,R) RL.append(point_to_node[r] = G.new_node(r));

cout << "\n\ntime to create 2n random points and 2n nodes = " << used_time(T);

int d = dclock();

cout << "\n\n";

 { ⟨solution with pricing⟩ }

cout << "\n\ntime with pricing = " << used_time(T);
cout << "\n\nwall time         = " << dclock() - d;

if (n <= 400)
{

T = used_time(); d = dclock();

cout << "\n\n";

 {⟨solution without pricing⟩}

cout << "\n\ntime for solution without pricing = " << used_time(T);
cout << "\n\nwall time                         = " << dclock() - d;
}

cout << "\n\n";
  }
}
```

The solution without pricing is trivial. We construct the complete bipartite graph and compute the minimum cost perfect matching. Observe that I round distances to integers (after multiplying with a large number); otherwise, there is the danger of round-off error in the computation of node potentials.

⟨*solution without pricing*⟩≡

```
edge e;
forall_edges(e,G) G.del_edge(e);

forall(b,B)
forall(r,R)
{ G.new_edge(point_to_node[r],point_to_node[b],distance(b,r)); }

list<edge> M = MIN_WEIGHT_ASSIGNMENT(G,G.edge_data());
double sum = 0;
forall(e,M) sum += G[e];
cout <<"\nweight of matching = " << sum;

forall_edges(e,G) G.del_edge(e);
```

In the solution with pricing we work on a sparse graph. We connect every node to its k nearest blue nodes. The LEDA data structure *point_set* allows us to search for nearest neighbors. It comes handy here. I also add an edge connecting the i-th red node with the i-th blue node; this is to make sure that there is a perfect matching.

We then compute a minimum cost perfect matching and also the potential function proving optimality. For every edge $(r, b) \in G$ we have $dist(r, b) \geq pot(r) + pot(b)$. It is the task of pricing to find out whether this holds for all pairs of red and blue point.

⟨*solution with pricing*⟩≡

```
edge e;
forall_edges(e,G) G.del_edge(e);

float T = used_time();

int k = 10; ;
cout << "\n\ninitial graph consists of three kinds of edges: ";
cout << "the edges in a trivial perfect matching, namely edges (ri,bi) for ";
cout << "an arbitrary numbering of the points. ";
cout << "Also the k nearest neighbors of each red and each blue point; k = " << k;
cout << "\n\ninitial number of edges = " << n*(2*k+1);

point_set BS(B);  // a point set for the blue points
point_set RS(R);  // a point set for the red points

list_item b_item = B.first();

forall(r,R)
{ G.new_edge(point_to_node[r],point_to_node[B[b_item]],distance(b,r));
  b_item = B.succ(b_item);

  list<node> BN = BS.nearest_neighbors(r,k);
  node v;
  forall(v,BN)
  { point b = BS.pos(v);
    node vr = point_to_node[r], vb = point_to_node[b];
    G.new_edge(vr,vb,distance(b,r));
```

```
    }
  }

  forall(b,B)
  { list<node> RN = RS.nearest_neighbors(b,k);
    node v;
    forall(v,RN)
    { point r = RS.pos(v);
      node vr = point_to_node[r], vb = point_to_node[b];
      G.new_edge(vr,vb,distance(b,r));
    }
  }

  cout << "\n\ntime to set up the initial graph = " << used_time(T);

  while(true) {

  bool done = true;

  node_array<double> pot(G);

  list<edge> M = MIN_WEIGHT_ASSIGNMENT(G,G.edge_data(),pot);

  cout << "\n\ntime to compute optimal assignment = " << used_time(T);

  int m = G.number_of_edges();

  ⟨pricing⟩

  cout << "\n\ntime for pricing = " << used_time(T);

  if ( done )
  { double sum = 0;
    edge e;
    forall(e,M) sum += G[e];
    cout << "\n\nsum = " << sum;
    break;
  }

  cout << "; Pricing added " << G.number_of_edges() - m <<" edges";

  }
```

We have computed a minimum weight assignment on the sparse graph; *pot* is a potential function proving optimality of the current matching. We need to check that there is no edge with negative reduced cost. Consider a pair (b, r) of blue and red node. We need to check that there is no pair with

$$dist(b, r) < y_b + y_r$$

We simply run over all pairs and check the inequality. If there are violations, we add for every red node up to K new edges. We add the edges with the K smallest values of $dist(b, r) - (y_b + y_r)$.

$\langle pricing \rangle \equiv$

```
int K = 10;

node rv,bv;

forall(rv,RL)  // iterate over all red nodes
{ point r = G[rv];
  list<node> BN;
  forall(bv,BL)
  { point b = G[bv];
    if ( distance(r,b) < pot[bv] + pot[rv])
      insert(BN,G,bv,rv,pot);

    if ( BN.length() > K ) BN.pop_back(); // delete last element
  }
  forall(bv,BN)
  { G.new_edge(rv,bv,distance(r,G[bv]));
    done = false;
  }
}
```

$\langle insert \rangle \equiv$

```
void insert(list<node>& BN, const GRAPH<point,double>& G,
            node bv, node rv,
            const node_array<double>& pot)
{ if ( BN.empty() ) { BN.push(bv); return; }
  point r = G[rv], b = G[bv];
  double D = distance(r,b) - pot[bv] - pot[rv];
  list_item it;
  forall_items(it,BN)
  { node w = BN[it];
    if ( D <= distance(r,G[w]) - pot[rv] - pot[w] ) break; }
  // insert b before it (at end of list if it == nil
  if ( it == nil ) BN.append(bv);
  else BN.insert(bv,it,LEDA::before);
}
```

An alternative strategy is as follows (I conjecture that this strategy is much better): We have a current matching M. In the next round we use the edges in M plus for each red point r the K edges with the lowest $dist(r,b) - y_r - y_b$ values. Similarly for the blue points. Observe that we used this strategy in the initialization (with all potentials equal to zero). Who implements this strategy for me?

We may also use geometric data structures to find the candidate pairs. If $dist(r,b) < y_b + y_r$ then either

$$dist(b,r) < 2 * y_b \quad \text{or} \quad dist(b,r) < 2 * y_r$$

We check for the existence of such a pair by performing circular range queries around b and r. The following piece of code worked with an earlier version of the program. It does not work anymore.

⟨*pricing_complex*⟩≡

```
forall(r,R)
{ // construct a circle with radius y_r
  double yr = pot[point_to_node[r]];
  if ( yr <= 0 ) continue;
  circle C(r,2*yr));
  list<node> BN = BS.range_search(C);
  if ( BN.length() > 0 )
  { node v;
    forall(v,BN)
    { point b = BS.pos(v);
      if ( distance(r,b) < pot[point_to_node[r]] + pot[point_to_node[b]])
      { k_large_enough = false; break; }
    }
  }
}
if ( k_large_enough )
{ forall(b,B)
  { // construct a circle with radius y_b
    double yb = pot[point_to_node[b]];
    if ( yb <= 0 ) continue;
    circle C(b,2*yb));
    list<node> RN = RS.range_search(C);
    if ( RN.length() > 0 )
    { node v;
      forall(v,RN)
      { point r  = RS.pos(v);
        if ( distance(b,r) < pot[point_to_node[r]] + pot[point_to_node[b]])
        { k_large_enough = false; break; }
      }
    }
  }
}
```

Experimental Results: In the first experiment I compare the two solutions. For $n = 400$, the two solutions have about the same running time. The complete graph has 160000 edges.

```
number of nodes = 400
```

```
time for pricing = 0.34
time with pricing = 5.85
wall time         = 6
time for solution without pricing = 8.54
wall time                         = 9
```

For $n = 1600$ the situation changes completely. The complete graph has now $(2^4 10^2)^2 = 256 \cdot 10^4 = 2.5 \cdot 10^6$ edges. This implies a job size of 350 Megabytes. My workstation has 512 Megabytes of main memory but according to "top" a large fraction of it is used for netscape, the window manager, ... Thus the job had to swap out part of the main memory. This takes time which is visible in the elapsed time.

```
number of nodes = 1600
time with pricing = 125.96
wall time         = 132
time for solution without pricing = 190.35
wall time                         = 462
```

I next analyze the solution with pricing. Here is the statistics for $n = 3200$. We observe that we go through several rounds and that in early rounds we add a significant number of edges. It would therefore make sense to start with a larger number of initial edges. Say set $k = 30$. We also observe that the cost of pricing is significant. In fact, the cost of pricing is larger than the cost of computing the optimal assignment. It would therefore be worthwhile, to use more sophisticated techniques in the pricing step (circular range queries). Also, we might improve the (re)computation of the perfect matching as follows.

Assume that we add a set E' of edges. We do the following.

1. For every node incident to an edge in E' we change its potential so as to make sure that $dist(r, b) \geq y_r + y_b$ for all edges in E'. This is easy to achieve by simply reducing the dual values appropriately. For example, by setting $y_r = dist(r, b) - y_b$

2. We unmatch all nodes for which we changed the potential in the first step.

3. Observe that matching edges which are not incident to a node whose potential has changed are still tight. We use the matching and potential function obtained in this way to initialize the perfect matching algorithm.

It might also be worthwhile to remove edges with $dist(r, b) \gg y_r + y_b$.

```
number of nodes = 3200
time to create 2n random points and 2n nodes = 0.0500031

I start with edges to k nearest neighbors plus one edge; k = 5
initial number of edges = 19200
time to set up the initial graph = 1.87
```

```
time to compute optimal assignment = 5.73001
time for pricing = 49.39; Pricing added 15984 edges
time to compute optimal assignment = 8.27
time for pricing = 46.98; Pricing added 15617 edges
time to compute optimal assignment = 10.14
time for pricing = 42.19; Pricing added 15325 edges
time to compute optimal assignment = 13.29
time for pricing = 40.4; Pricing added 14279 edges
time to compute optimal assignment = 16.36
time for pricing = 35.12; Pricing added 13300 edges
time to compute optimal assignment = 21.11
time for pricing = 27.74; Pricing added 9737 edges
time to compute optimal assignment = 24.17
time for pricing = 26.12; Pricing added 5290 edges
time to compute optimal assignment = 26.39
time for pricing = 25.82; Pricing added 2374 edges
time to compute optimal assignment = 25.47
time for pricing = 25.6299; Pricing added 751 edges
time to compute optimal assignment = 26.4
time for pricing = 25.68; Pricing added 114 edges
time to compute optimal assignment = 26.4
time for pricing = 25.61; Pricing added 7 edges
time to compute optimal assignment = 26.84
time for pricing = 25.68
sum = 1.41577e+09
time with pricing = 628.84
```

Here is the effect of starting with $\sqrt{n}$ edges in the initial graph.

```
number of nodes = 3200
time to create 2n random points and 2n nodes = 0.0400009

I start with edges to k nearest neighbors plus one edge; k = 56
initial number of edges = 182400
time to set up the initial graph = 4.96
time to compute optimal assignment = 37.86
time for pricing = 23.15; Pricing added 1654 edges
time to compute optimal assignment = 38.85
time for pricing = 23.13; Pricing added 407 edges
```

```
time to compute optimal assignment = 39.16
time for pricing = 23.19; Pricing added 18 edges
time to compute optimal assignment = 40.04
time for pricing = 23.35; Pricing added 1 edges
time to compute optimal assignment = 39.1
time for pricing = 23.12
sum = 1.5477e+09
time with pricing = 315.95
wall time         = 332
```

The solution with pricing uses 65 Mbytes for $n = 6400$. There is lots of room for improvement.

1.2 Diplomarbeitsthema

Efficient Pricing Strategies for Geometric Instances of Matching Problems

Geometric instances of matching problems lead to problems on complete (bipartite) graphs and hence the space requirement of naive solutions grows quadratically in the problem size. This puts a severe limit on the size of problems that can be solved by the naive approach.

Pricing can make a big difference here. Efficient realization of pricing requires to combine techniques from combinatorial optimization and computational geometry.

In this thesis

- we first describe the theoretical background for pricing in matching problems. We put equal emphasis (and this is the novel aspect of the thesis) on combinatorial optimization and computational geometry. This part of the thesis will require you to read and to evaluate literature in combinatorial optimization and computational geometry. It is the theoretical part of the thesis.

- we als describe implementations for bipartite matching (matching n red with n blue points) and general matching (matching $2n$ points). The implementation of general matching extends the implementation described in [MS00, Sch00]. Pricing for the general matching problem is discussed in [CCPS98, pages 159–161]

References

[CCPS98] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization.* John Wiley & Sons, Inc, 1998.

[MS00] K. Mehlhorn and G. Schäfer. Implementation of $O(nmlogn)$ weighted matchings in general graphs: The power of data structures. In *Workshop on Algorithm Engineering (WAE)*, Lecture Notes in Computer Science, to appear, 2000. `www.mpi-sb.mpg.de/~mehlhorn/ftp/WAE00.ps.gz`.

[Sch00] G. Schäfer. Weighted matchings in general graphs. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2000.