Algorithm Design and Analysis(H)
Southern University of Science and Technology
Mengxuan Wu
12212006

Assignment 5
Kuhn-Munkres Algorithm for Min-Cost Perfect Matching

Mengxuan Wu

1 Psuedocode

Algorithm 1: Kuhn-Munkres

```
Input: $costMatrix$, $n$
Output: Minimum cost of the perfect matching
Initialize $matchedJob$ to all -1;
Initialize $labelWorker$ to all 0;
Initialize $labelJob$ to all 0;

for $worker \leftarrow 1$ to $n$ do
    for $job \leftarrow 1$ to $n$ do
        if costMatrix[worker][job] > labelJob[job] then
            labelJob[job] $\leftarrow$ costMatrix[worker][job];
        end
    end
end
for $worker \leftarrow 1$ to $n$ do
    bfsMatch(worker, costMatrix, n, labelWorker, labelJob, matchedJob);
end
minCost $\leftarrow$ 0;
for $job \leftarrow 1$ to $n$ do
    if matchedJob[job] != -1 then
        minCost $\leftarrow$ minCost + costMatrix[matchedJob[job]][job];
    end
end
return $minCost$;
```

Algorithm 2: BFS Match

```
Input: worker, $costMatrix$, $n$, $labelWorker$, $labelJob$, $matchedJob$
Output: Updates $matchedJob$ to reflect the minimum cost matching
Initialize $visitedJob$ to all false;
currentJob ← 0;
matchedJob[currentJob] ← worker;
for $i \leftarrow 1$ to $n$ do
    slack[i] ← $\infty$;
end
while true do
    currentWorker ← matchedJob[currentJob];
    delta ← $\infty$;
    visitedJob[currentJob] ← true;
    for $job \leftarrow 1$ to $n$ do
        if visitedJob[job] then
            Continue;
        end
        newSlack ← costMatrix[currentWorker][job] - labelWorker[currentWorker]
         - labelJob[job];
        if newSlack < slack[job] then
            slack[job] ← newSlack;
            predecessor[job] ← currentJob;
        end
        if slack[job] < delta then
            delta ← slack[job];
            nextJob ← job;
        end
    end
    for $i \leftarrow 0$ to $n$ do
        if visitedJob[i] then
            labelWorker[matchedJob[i]] ← labelWorker[matchedJob[i]] + delta;
            labelJob[i] ← labelJob[i] - delta;
        end
        else
            slack[i] ← slack[i] - delta;
        end
    end
    currentJob ← nextJob;
    if matchedJob[currentJob] == -1 then
        Break;
    end
end
while currentJob do
    matchedJob[currentJob] ← matchedJob[predecessor[currentJob]];
    currentJob ← predecessor[currentJob];
end
```

2 Core Idea

2.1 Problem Statement

Given a bipartite graph $G = (U, V, E)$, where U and V are the two disjoint sets of vertices and E is the set of edges between them. Each edge $e \in E$ has a cost $c(e)$. The goal is to find a perfect matching that minimizes the total cost of the edges in the matching.

The pseudocode above describes the Kuhn-Munkres algorithm, which solves this problem in $O(n^3)$ time complexity. Also, a min-cost perfect matching can be found by negating the costs of the edges and running the algorithm. We will show the correctness of this transformation in the following sections.

2.2 Core Idea

The Kuhn-Munkres algorithm's core idea is give each worker and job a label, and calculate the reduced cost of each edge accordingly. Then, we try to find an augmenting path in the graph, which is a path that starts and ends with unmatched vertices, and the edges in the path alternate between matched and unmatched edges. To obtain the minimum cost matching, we try to follow the augmenting path with the minimum cost, and update the labels of the workers and jobs accordingly.

2.2.1 Label

In this section, we use a definition different from textbook but more widely used in programming competition. However, the two definitions are equivalent.

The labels are the key to the algorithm. And they do have economic interpretations. We can think of the cost of an edge as the maximum budget that we can afford to hire a worker for a job. The label of a worker, $p(u) \geq 0$, is the amount of money that we need to pay to hire the worker. The label of a job, $p(v) \leq 0$, is the reward of the job. The reduced cost of an edge is $r(u, v) = c(u, v) - p(u) - p(v)$.

Now we define a feasible labeling. A labeling is feasible if for all edges (u, v), $p(u) + p(v) \leq c(u, v)$ (equivalently, $r(u, v) \geq 0$) is satisfied. The economic interpretation is that the net cost of assigning a worker to a job (calculated as the cost of hiring the worker minus the reward of the job) should be within the budget. Furthermore, there is always a feasible labeling that turns all matching edges into tight edges (edges with $r(u, v) = 0$), while keeping all non-matching edges with $r(u, v) \geq 0$. The interpretation is that we want to hire the best worker for his/her assigned job, which will exploit the full budget.

2.2.2 Augmenting Path

The algorithm tries to find an augmenting path in the graph. An augmenting path is a path that starts and ends with unmatched vertices, and the edges in the path alternate between matched and unmatched edges. An alternative way is that every matched edge is directed from the job to the worker, and every unmatched edge is directed from the worker to the job.

Since the idea of this augmenting path is exactly the same as the augmenting path in the max-flow algorithm, we do not repeat explaining it here.

2.3 Critical Procedure

The critical procedure of the algorithm is to update the labels of the workers and jobs. The procedure is as follows:

1. Consider an augmenting path P starting from a worker u.
2. If no such path exists, recalculate the labels of the workers and jobs.
3. As long as the path exists, propagate the matching along the path, go to step 1 with the next worker.

To satisfy the feasibility condition, we Initialize all labels to 0. Obviously, $r(u, v) = c(u, v)$ for all edges, and the feasibility condition is satisfied.

Then, at each step of the algorithm, we try to increase the cardinality of the matching by 1. This is done by finding an augmenting path with the Hungarian method. Then we update the labels of the workers and jobs to maintain the feasibility condition while making the new matching edge tight.

The update requires finding a minimum Δ (to be noticed that Δ is negative):

$$\Delta = \min_{i \in Z_1, j \notin Z_2} c(i, j) - p(i) - p(j)$$

where Z_1 is the set of workers that have been selected, and Z_2 is the set of jobs that have not been selected in the augmenting path. Then we update the labels as follows:

$$\begin{aligned} p(i) =& p(i) + \Delta \quad \text{for all } i \in Z_1 \\ p(j) =& p(j) - \Delta \quad \text{for all } j \in Z_2 \end{aligned}$$

After the update, the old matching edges remain tight, and the new matching edge becomes tight.

3 Proof of Correctness

Such update of the labels is guaranteed to maintain the feasibility condition.

- For $i \in Z_1$ and $j \in Z_2$, $r(i, j)$ remains unchanged since the Δ cancels out.
- For $i \notin Z_1$ and $j \in Z_2$, the reduced cost $r'(i, j) = r(i, j) - \Delta$, the reduced cost increases and the inequality holds.
- For $i \in Z_1$ and $j \notin Z_2$, since we choose the minimum Δ, the reduced cost decreases, but it is still non-negative.
- For $i \notin Z_1$ and $j \notin Z_2$, the reduced cost remains unchanged.

Since the Δ is chosen to be the minimum, the particular edge that produce the minimum Δ becomes tight after the update. And the tight edges of the old matching remain tight. Hence, the cardinality of $|Z_1| + |Z_2|$ strictly increases.

Since the new matching may replace all the old matching edges, and each recalculation of the labels strictly increases the cardinality of the matching, the algorithm at most recalculate the labels n times (since the cardinality of the matching is at most n). This will terminate the while loop in the BFS Match procedure. Hence, we can see the termination is guaranteed.

4 Complexity Analysis

4.1 Time Complexity

The algorithm calls the BFS Match procedure n times, and each call of the BFS Match procedure recalculate the labels at most n times, and each recalculation of the labels takes $O(n)$ time. Hence, the total time complexity is $O(n^3)$.

4.2 Space Complexity

The main space complexity of the algorithm is the cost matrix, which takes $O(n^2)$ space. All auxiliary arrays take $O(n)$ space. Hence, the total space complexity is $O(n^2)$.

5 Extension to max-cost perfect matching

To find the max-cost perfect matching, we can negate the costs of the edges and run the algorithm. However, this may cause the reduced costs to be negative at the beginning. Hence, we add a constant c to all the costs to make all the reduced costs non-negative.

The correctness of this transformation is guaranteed by two facts:

1. If we negate the costs of the edges, the min-cost perfect matching becomes the max-cost perfect matching.
2. Adding a constant to all the costs does not change the optimal solution.

For fact 1, we can prove with contradiction. If the original min-cost perfect matching M is not the max-cost perfect matching after negating the costs, then there exists a new max-cost perfect matching M' that has a higher cost than M. Then we revert the negation of the costs, and we find that M' has a lower cost than M, which contradicts the fact that M is the min-cost perfect matching.

For fact 2, since perfect matching always contain the same number of edges, adding a constant c to all the costs will equally add $n \cdot c$ to the total cost of the all perfect matchings. Hence, the optimal solution remains the same.