MANCHESTER
1824

The University of Manchester

An algorithm for computing the eigenvalues of a max-plus matrix polynomial

James, Hook

2016

MIMS EPrint: **2016.44**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

An algorithm for computing the eigenvalues of a max-plus matrix polynomial

James Hook

School of Mathematics, The University of Manchester, Manchester, M13 9PL, UK.
james.hook@manchester.ac.uk,

Abstract

Max-plus matrix polynomial eigenvalues provide a useful approximation to the order of magnitude of the eigenvalues of a classical (i.e. real or complex) matrix polynomial. In this paper we review the max-plus matrix eigensolver of Gassner and Klinz [1] and present our extension of this algorithm to the max-plus matrix polynomial case. Our max-plus matrix polynomial algorithm computes all nd max-plus eigenvalues of a $n \times n$ degree d max-plus matrix polynomial with worst case cost $O(n^3 d)$ in the dense case, which is the best that we are aware of.

Keywords:

1. Introduction

Max-plus algebra concerns the max-plus semiring $\mathbb{R}_{\max} = \mathbb{R} \cup \{-\infty\}$ with binary operations max and plus

$$a \oplus b = \max\{a, b\}, \quad a \otimes b = a + b, \quad \text{for all } a, b \in \mathbb{R}_{\max}.$$

Recently max-plus algebra has attracted interest from the numerical linear algebra community as a useful way of approximating difficult classical numerical linear algebra problems. The idea is that a classical numerical linear algebra problem can be transformed into a max-plus one, which is easier to solve. The solution to the max-plus problem can then be used to assist in the solution of the original problem. For example the nd eigenvalues of an $n \times n$ degree d complex matrix polynomial can be approximated using the d max-plus roots of a degree d max-plus scalar polynomial. This approximation of the eigenvalues can then be used in the deign of scaling stratergies for the original matrix polynomial, which can dramatically improve the accuracy of the subsequent eigenvalue computation for certain tough problems [2, 3, 4].

A possible refinement of this technique is to transform the $n \times n$ degree d matrix polynomial into an $n \times n$ degree d max-plus matrix polynomial. We can then approximate the eigenvalues of the standard matrix polynomial with those of the max-plus matrix polynomial. As before these approximated eigenvalues can be used in the design of scailngs for the original matrix polynomial, only using this more sophisticated max-plus approximation could lead to superior results over a wider range of problems [5]. Other applications of max-plus eigenvalues in linear algebra include approximation of matrix singular values and condition numbers [6], as well as computing the asymptotics of the eigenvalues of certain perturbed systems [7].

In order for such approximations to be useful in practice it is essential that the cost of computing the max-plus approximations is no more than the cost of computing the quantities of interest directly. All d max-plus roots of a degree d max-plus scalar polynomial p can be computed with cost $O(d)$ using the Graham scale algorithm [8]. In [1] Gassner and Klinz present an algorithm which computes all n max-plus eigenvalues of an $n \times n$ max-plus matrix $A \in \mathbb{R}_{\max}^{n\times n}$ with worst case cost $O\big(n\tau_A + n^2\log(n)\big)$, where τ_A is the number of finite entries in A (finite, i.e. non $-\infty$, entries play the role of non-zero entries in max-plus algebra). This algorithm is itself based on a parametric shortest path algorithm of Young, Tarjan and Orlin [9]. In [10] Burkard and Butkovic present an alternative algorithm which computes all n max-plus eigenvalues of an $n \times n$ max-plus matrix with worst case cost $O\big(n^2\tau_A + n^2\log(n)\big)$. This algorithm is adapted to the matrix polynomial case by Sharify in [11] to provide an algorithm which computes all nd eigenvalues of an $n \times n$ degree d max-plus matrix polynomial $P \in \mathbb{R}_{\max}[x]^{n\times n}$ with worst case cost $O\big(n^2 d\tau_P + n^2 d\log(n)\big)$, where τ_P is the number of entries in P which are not identically $-\infty$.

In this paper we review the max-plus single matrix eigenvalue algorithm of Gassner and Klinz and present our extension of this algorithm to the matrix polynomial case. Our algorithm has worst case cost $O\big(nd\tau_P + n^2 d\log(n) + \tau_P d\log(d)\big)$, which is the best that we are aware of. Note that in the dense case our algorithm has cost $O(n^3 d)$, which is a factor d cheaper than than for the classical case.

The remainder of this paper is organized as follows. In subsection 1.1 we review the Graham scan algorithm for computing the roots of a max-plus scalar polynomial. In section 2 we review the max-plus single matrix eigenvalue algorithm of Gassner and Klinz. In section 3 we present our max-plus matrix polynomial eigenvalues algorithm.

1.1. Polynomial roots

A degree-d *max-plus scalar polynomial* is a function of the form

$$p(x) = \bigoplus_{k=0}^{d} p_k \otimes x^{\otimes k} = \max_{k=0}^{d} p_k + kx,$$

where $p_0, \dots, p_d \in \mathbb{R}_{\max}$. Max-plus scalar polynomial are convex piecewise affine functions. The *max-plus roots* of a max-plus scalar polynomial p are the values of x at which it is non-differentiable. The *multiplicity* of a root r is given by the jump in derivative at that point. The sum of the multiplicities of the roots of a polynomial p is equal to its degree.

In this section we show how the Graham scan algorithm can be used to compute all d roots $r_1, \dots, r_d$ of a degree-d max-plus scale polynomial with cost $O(d)$.

Let $H(N)$ be the upper convex hull of the Newton polytope $N = \{(k, p_{d-k}) : k = 0, \dots, d\}$. The roots of $p(x)$ are given by the slopes of the line segments that make up H, and the multiplicity of a root r is given by the horizontal width of the corresponding line segment.

The Graham scan algorithm can compute the convex hull of a set of d points in the plane with cost $O\big(d\log(d)\big)$ [**?**]. However, we are given the points that make up N in increasing order of their first co-ordinate and we can exploit this extra information to speed things up slightly, see Algorithm 1.

We can think of the Graham scan algorithm as producing a sequence of upper convex hulls for the subsets $N(1), \dots, N(d) \subset N$ with

$$N(m) = \{(k, p_{d-k}) : k = 0, \dots, m\}.$$

At each stage we consider extending$H\big(N(m)\big)$ to contain $(m+1, p_{d-m-1})$ by adding a line segment connecting $(m+1, p_{d-m-1})$ to the previously included vertex (m, p_{d-m}). If the resulting hull is convex then we move onto the next stage, otherwise we remove vertices from the hull until it is convex. Thus each vertex is added once and removed at most once and the total number of operations performed by the algorithm is $O(d)$.

Algorithm 1 Max-plus scalar polynomial root finder

```
1: procedure BEGIN
2:     for $k = 1, \dots, d$ do
3:         $B(k) = k - 1$, $R(k) = p_{d-k} - p_{d-k+1}$,
4:         while $R(k) > R\big(B(k)\big)$ do
5:             $B(k) = B\big(B(k)\big)$, $R(k) = \frac{p_{d-k} - p_{d-B(k)}}{k - B(k)}$,
6:     $k = d$,
7:     while $k > 0$ do
8:         $R(k)$ a root of multiplicity $k - B(k)$,
9:         $k = B(k)$,
```

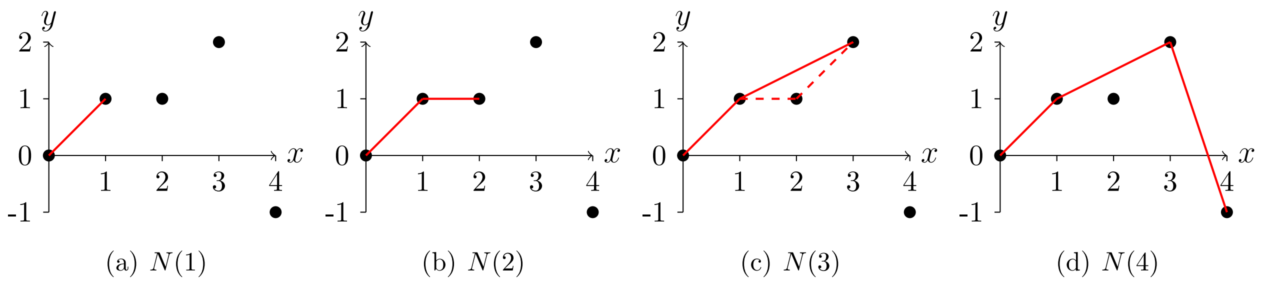


Figure 1: Convex hulls produced by Graham scan algorithm

Example Consider the max-plus scalar polynomial

$$p(x) = x^{\otimes 4} \oplus 1 \otimes x^{\otimes 3} \oplus 1 \otimes x^{\otimes 2} \oplus 2 \otimes x \oplus -1 = \max\{4x, 3x+1, 2x+1, x+2, -1\}.$$

The sequence of convex hulls produced in the course of Algorithm 1 are displayed in Figure 1. The upper convex hull of N consists of a line segment of slope one with width one, a line segment of slope a half with width two and a line segment with slope minus three and width 3. The roots of p are therefore given by $x = 1$, which is a simple root, $x = 0.5$, which is a double root and $x = -3$, which is a simple root.

As in the classical case, a polynomial can be factorized according to its roots

$$p(x) = (x \oplus 1) \otimes (x \oplus 0.5)^{\otimes 2} \otimes (x \oplus -3) = \max\{x, 1\} + 2\max\{x, 0.5\} + \max\{x, -3\}.$$

2. Matrix eigenvalues

In this section we review the max-plus single matrix eigenvalue algorithm of Gassner and Kilnz [1]. The *max-plus eigenvalues* $\mu_1, \ldots, \mu_n$ of a max-plus matrix $A \in \mathbb{R}_{\max}^{n\times n}$ are the max-plus roots of the max-plus scalar polynomial

$$\chi_A(x) = \operatorname{perm}(A \oplus x \otimes I) = \max_{\pi\in\Pi_n} \sum_{i=1}^{n} [A \oplus x \otimes I]_{i\pi(i)}, \tag{1}$$

which we call the *characteristic polynomial* of A, where $I \in \mathbb{R}_{\max}^{n\times n}$ is the max-plus identity matrix with zeros on the diagonal and minus infinity entries off of the diagonal and Π_n is the set of all permutations on $\{1, \ldots, n\}$. We also include $-\infty$ as an eigenvalue of multiplicity k, whenever the first k coefficients of χ_A are $-\infty$.

Computing the max-plus eigenvalues of $A \in \mathbb{R}_{\max}^{n\times n}$ is equivalent to computing the switching points of the following *parametric maximal matching problem.*

Let G_A be be the bipartite graph with left vertices $U = \{u(1), \ldots, u(n)\}$, right vertices $V = \{v(1), \ldots, v(n)\}$, constant edges $E(0) = \{e(0)_{ij} : u(i) \mapsto v(j)$, with weight $w(e(0)_{ij}) = a_{ij}$, for $i, j = 1, \ldots, n$ such that $a_{ij} \neq -\infty\}$ and parametric edges $E(1) = \{e(1)_i : u(i) \mapsto v(i)$, with weight $w(e(1)_{ij}) = x$, for $i = 1, \ldots, n\}$.

A *matching* or assignment of G_A is a subset of edges $M \subset E(0) \cup E(1)$, such that each vertex in U and V is incident to exactly one edge in M. Therefore, the edges in a matching M can be uniquely represented as $M = M(\pi, s) = \{(e(s_i)_{i\pi(i)} : i = 1, \ldots, n\}$, for some $s \in \{0, 1\}^n$ and some $\pi \in \Pi_n$. We say that the matching $M(\pi, s)$ *matches* the vertex $u(i)$ to the vertex $v\big(\pi(i)\big)$ with the edge $e(s_1)_{i\pi(i)}$. The weight of a matching is simply the sum of its edge weights

$$W\big(M(\pi, s)\big) = \sum_{k=1}^{n} w\big(e(s_k)_{i,\pi(i)}\big).$$

Note that

$$\chi_A(x) = \max_{\pi\in\Pi_n} \max_{s\in\{0,1\}^n} \sum_{k=1}^{n} w\big(e(s_k)_{i,\pi(i)}\big) = \max_{M} W(M), \tag{2}$$

where the rightmost maximum is taken over all matchings of G_A. Since the weights of the parametric edges depend on x, the weights of the matchings will also depend on x. The values of x where the matching that attain the maximum in (2) switches are the non-differentiability points of $\chi_A(x)$, which are the finite max-plus eigenvalues of A.

Given a matching M of G_A we define the *residual bipartite graph* $R_A(M)$ to be the graph obtained by taking G_A and reversing the direction of and minusing the weight of all of the edges in M. We call the left to right edges the *forwards edges* and the right to left edges the *backwards edges*. Now let C be a cycle in $R_A(M)$, we *augment* M with respect to C by taking the symmetric difference $M \triangle C = (M \cup C)/(M \cap C)$.

Lemma 2.1 *Let M be a matching of G_A and let C be a cycle in $R_A(M)$ then $M' = M \triangle C$ is also a matching of G_A and $W(M') = W(M) + W(C)$.*

Proof Each vertex in U and V is incident to exactly one edge in M. If $u(i)$ is a vertex not visited by C then M' will contain exactly one edge incident to $u(i)$. If $u(i)$ is a vertex visited by C then C will contain two edges incident to $u(i)$, one backwards edge which is also in M and one forwards edge which is not in M. When we augment M with respect to C we replace the former with the later to obtain M' so that $u(i)$ is incident to exactly one edge in M'. Likewise for the V vertices. $M' = M \triangle C$ is therefore a matching of G_A.

For the second part it is important to note that the weight of the matchings M and M' are calculated using the edge weights of the graph G_A but that the weight of the cycle C is calculated using the edge weights of the graph $R_A(M)$ in which some edges have had their weights minused. Then from the previous argement we have $W(M') - W(M)$ is equal to the sum of the weights of the forwards edges in C minus the sum of the unadjusted weights (i.e. their weights in G_A not $R_A(M)$) of the backwards edges in C. However since the backwards edges in $R_A(M)$ have all had their weights minused we have $W(M') - W(M)$ is equal to the sum of the weights (in $R_A(M)$) of all of the edges in C, which is equal to $W(C)$.

Lemma 2.2 *Let M and M' be matchings of $G_A(M)$ then there exists disjoint cycles $C_1, \ldots, C_k$ in $R_A(M)$ such that*

$$M \triangle (C_1 \cup \ldots \cup C_k) = M',$$

and $W(M') = W(M) + W(C_1) + \ldots + W(C_k)$.

Proof Let $M = M(\pi, s)$ and $M' = M'(\varpi, r)$. Consider $M \triangle M'$. If $u(i)$ is incident to the same edge in M as in M' then $M \triangle M$ will contain no edges incident to $u(i)$, likewise for any V vertices. Now suppose that $u(i)$ is incident to the edge $e(s_i)_{i\pi(i)}$ in M and $e(r_i)_{i\varpi(i)}$ in M' and that these edges are different. Then $u(i)$ must be incident to exactly two edges in $R_A(M)$, the forwards edge $e(r_i)_{i\varpi(i)}$, which is an edge from M and has therefore been reversed in the residual graph, and the backwards edge $e(s_i)_{i\pi(i)}$, which has not been reversed. Likewise any vertex $v(i)$ that is matched differently by M and M' will be incident to exactly two edges in $R_A(M)$, one forwards edge and one backwards edge. The set $M \triangle M'$ is such that each vertex in U and V is incident to either zero or two of its edges. This set must therefore consists of a finite number k of disjoint cycles $C_1, \ldots, C_k$ as any other structure would have vertices incident to one edge or more than two edges. Thus $M \triangle M' = C_1 \cup \ldots \cup C_k$.

Using the fact that $A \triangle B = C$ if and only if $A \triangle C = B$, we have $M \triangle (C_1 \cup \ldots \cup C_k) = M'$. Since the cycles are disjoint we have

$$M \triangle (C_1 \cup \ldots \cup C_{i+1}) = \big(M \triangle (C_1 \cup \ldots \cup C_i)\big) \triangle C_{i+1}.$$

So that from Lemma 2.1 we have

$$W\big(M \triangle (C_1 \cup \ldots \cup C_{i+1})\big) = M \triangle (C_1 \cup \ldots \cup C_i) + W(C_{i+1}),$$

and therefore $W(M') = W(M) + W(C_1) + \ldots + W(C_k)$.

Corollary 2.3 *Let M be a matching of G_A then M attains the maximum in (2) for $x \in [a, b]$ if and only if $R_A(M)$ contains no positively weighted cycles for $x \in [a, b]$.*

Proof Suppose that M attains the maximum in (2). Let C be a cycle in $R_A(M)$, then from Lemma 2.1 $M' = M \triangle C$ is also a matching with weight $W(M') = W(M) + W(C)$. But if M attains the maximum in (2) then $W(M') \leq W(M)$ so $W(C) \leq 0$. Therefore $R_A(M)$ contains no positively weighted cycles.

Now suppose that $R_A(M)$ contains no positively weighted cycles. Let M' be a matching of G_A, then from Lemma 2.2 there exists disjoint cycles $C_1, \ldots, C_k$ in $R_A(M)$ such that $M \triangle (C_1 \cup \ldots \cup C_k) = M'$ and $W(M') = W(M) + W(C_1) + \ldots + W(C_k)$. But if $R_A(M)$ contains no positively wighted cycles then $W(M') \leq W(M)$. Therefore M attains the maximum in (2).

Gassner and Klinz's max-plus eigensolver algorithm works as follows. Suppose that we already know the maximally weighted matching M for some value $x = x_1$. We detect the greatest value $x_2 \leq x_1$ at which a positively weighted cycle C in $R_A(M)$ will emerge. This will be a cycle C with $W(C) = 0$ for $x = x_2$ and $W(C) > 0$ for $x < x_2$. We argument M with respect to C to obtain $M' = M \triangle C$. Now M' is the maximally weighted matching for $x = x_2$. We continue finding further emergent positively weighted cycles and reducing x. By maintaining the property that $R_A(M)$ contains no positively weighted cycles we ensure that we always have the maximally weighted matching (due to Corollary 2.3). The values of x at which the maximally weighted matching changes are the eigenvalues of A and the multiplicity of an eigenvalue is given by the change in the parametric weight of the maximal matching at that eigenvalue.

Example Consider the max-plus matrix

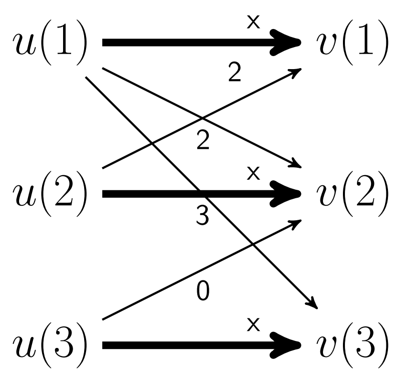
$$A = \begin{bmatrix} -\infty & 2 & 3 \\ 2 & -\infty & -\infty \\ -\infty & 0 & -\infty \end{bmatrix}.$$

Expanding the characteristic polynomial of A yields

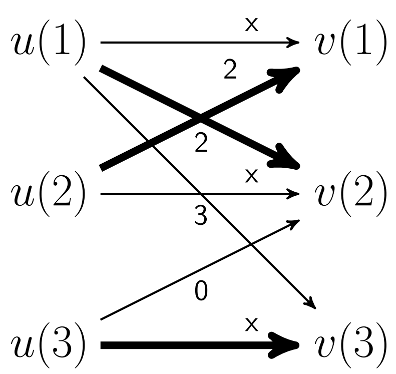
$$\chi_A(x) = x^{\otimes 3} \oplus 4 \otimes x \oplus 5 = \max\{3x, 4 + x, 5\}.$$

The roots of χ_A and therefore the eigenvalues of A are given by $\mu_1 = 2$, which is a double root (where the derivative jumps by 2) and $\mu_2 = 1$, which is a simple root.

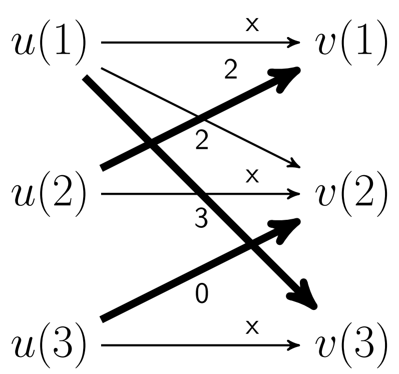
The three different matchings that attain the maximum in (2) are displayed in Figure 2 subfigures a,b,c. Note that the intervals in which a particular matchings is maximally weighted is always bounded by a pair of max-plus eigenvalues of A. Figure 2 subfigure d displays the graph $R_A(M_1)$, this residual bipartite graph contains two cycles $u(3) \mapsto v(2) \mapsto u(2) \mapsto v(1) \mapsto u(1) \mapsto v(3) \mapsto u(3)$, which has weight $-x + 0 - x + 2 - x + 3 = 5 - 3x$ and $C_1 = u(2) \mapsto v(1) \mapsto u(1) \mapsto v(2) \mapsto u(2)$, which has weight $-x + 2 - x + 2 = 4 - 2x$. The second cycle C_1 is the first to emerge with a positive weight at $x = 2$. Viewed as a subset of edges C_1 is given by $C_1 = \{e(1)_{11}, e(0)_{12}, e(1)_{22}, e(0)_{21}\}$. We augment M_1 with respect to C_1 to obtain $M_2 = M_1 \triangle C_1 = \{e(0)_{12}, e(0)_{21}, e(1)_{33}\}$. Likewise $C_2 = u(3) \mapsto v(2) \mapsto u(1) \mapsto v(3) \mapsto u(3)$ is the first positive weighted cycle to emerge in $R_A(M_2)$ and $M_3 = M_2 \triangle C_2$.



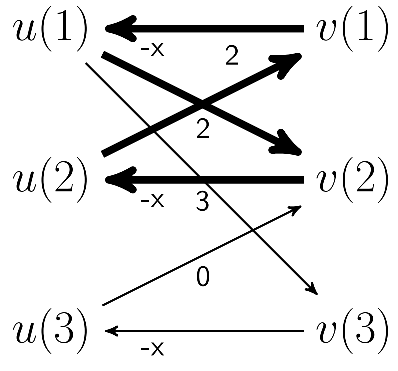
(a) For $x \in [2, \infty)$,
$M_1 = \{e(1)_{11}, e(1)_{22}, e(1)_{33}\}$.



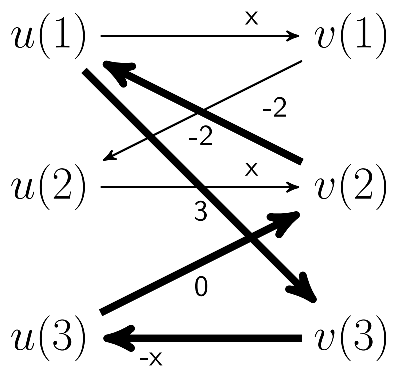
(b) For $x \in [1, 2]$
$M_2 = \{e(0)_{12}, e(0)_{21}, e(1)_{33}\}$.



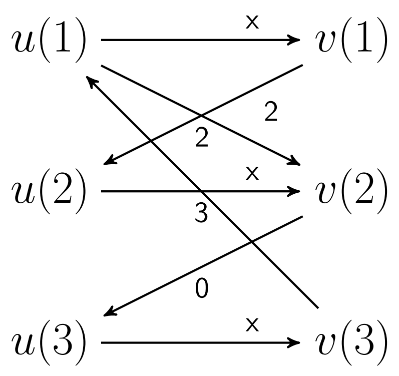
(c) For $x \in [-\infty, 1]$
$M_3 = \{e(0)_{13}, e(0)_{21}, e(0)_{32}\}$.



(d) $R_A(M_1)$.



(e) $R_A(M_2)$.



(f) $R_A(M_3)$.

Figure 2: a,b,c) Bipartite graph G_A, with maximally weighted matchings (bold edges). d,e,f) Residual graphs with augmenting cycles (bold edges).

Gassner and Klinz's algotithm detects the emergence of positively weighted cycles by maintaining a maximally weighted spanning tree T in $R_A(M)$. When M is the maximally weighted matching T exists, but as soon as a positively weighted cycle emerges T becomes ill defined. The algorithm starts by initializing $R_A(M)$ and T for a very large positive value of x then successively finds the next value of x at which T changes and updates T accordingly. When x crosses a value at which a positively wighted cycle emerges, i.e. a max-plus eigenvalue of the problem matrix, this is detected and stored and M is updated accordingly.

2.1. The algorithm

The following algorithm calculates all n eigenvalues of a max-plus matrix $A \in \mathbb{R}_{\max}^{n\times n}$ with cost $O\big(n\tau_A + n^2\log(n)\big)$, where τ_A is the number of finite entries in A

The algorithm we present is slightly different from the one presented by Gassner and Klinz, their algorithm workes by increasing the parameter x, whereas ours works by decreasing it. This reduces the cost of the initialization phase from $O(n\tau_A)$ to $O(\tau_A)$, where τ_A is the number of finite entries in the matrix, so although this does not affect the overall complexity of the algorithm, our version should be faster. Figure 3 is a flowchart which summarizes the algorithm's procedure.

1. We set $M_1 = E(1)$ and construct $R_A(M_1)$. Note that since M_1 consists of all of the parametric edges we have $W(M_1) = nx$ and there is no other matching with this parametric coefficient. Therefore for sufficiently large positive x, M_1 is the maximally weighted matching.
   Next we adjoin a root vertex r, with constant edges $E(r) = \{e(r)_{ri} : r \mapsto u(i)$, with weight $w(e(r)_{ri}) = 0$, for $i = 1, \dots, n\}$. This results in the graph $R_A(M_1) \cup r$.
   We then construct a maximally weighted spanning tree T through $R_A(M_1) \cup r$, rooted at r. The weight of the different trees are compared by assuming that x is very large and positive, so that the constant part of the weight is maximised subject to the parametric part being maximised first. Thus T consists of the edges $E(r)$ along with $\{e(0)_{m(j)j} : \ j = 1, \dots, n\}$, where $a_{m(j)j} = \max_i a_{ij}$, if the maximally weighted tree is not unique, then it does not matter which maximally weighted tree we choose.
   For a vertex $a \in U \cup V$ we define the *depth* $D(a)$ to be the sum of the edge weights along the path from r to a through T. The depth of

a vertex has a constant coefficient $D(a)_0$ and a parametric coefficient $D(a)_1$, so that $D(a) = D(a)_0 + xD(a)_1$. We will also refer to the constant and parameter coefficients of the edge weights in the same way so that $w(e) = w(e)_0 + xw(e)_1$ for all $e \in E(r) \cup E(0) \cup E(1)$. Now we compute the *edge keys* for each edge $e : a \mapsto b$ by

$$k(e) = \frac{D(a)_0 + w(e)_0 - D(b)_0}{D(b)_1 - D(a)_1 - w(e)_1}, \tag{3}$$

if this denominator is strictly positive and $k(e) = -\infty$ otherwise. Now $D(b)$, the depth of the vertex b, is the weight of the path from r to b through T and $D(a) + w(e)$ is the total weight of the path from r to b consisting of the path through T from r to a then the edge e from a to b. The edge key $k(e)$ is therefore the value of x at which the weight of this second path switches to being greater than the weight of the first. We define the *vertex key* $k(b)$ of a vertex b to be the maximum over of all vertex edge keys, for edges that end at vertex b. If e is the edge that corresponds to vertex b's key then we say that the *origin* of b's key is e. If this maximum is not unique, then it does not matter which maximal edge key we choose.

2. We take the maximum vertex key, $k(b)$ with origin e say. We call b the *pivot vertex* and $e : a \mapsto b$ the *pivot edge*. If this maximum is not unique, then it does not matter which maximal vertex key we choose. If there are no finite keys then we store minus infinity as an eigenvalue with multiplicity equal to n minus the sum of the multiplicities of the finite eigenvalues stored so far and terminate the algorithm.
3. We update T according to the pivot edge $e : a \mapsto b$. To do this we replace the current edge in T that edges at b with the new edge e. If T' contains a cycle C and we go to step 5, otherwise, if T is still a tree then we go to step 4.
4. We compute the keys for the new tree T'. It is not necessary to check all of the edge keys to do this. For every vertex c, downstream of the pivot vertex b in T, in the sense that there exists a directed path through T from b to c, we update the edge keys of any edges incident to c, then update the vertex keys from these edge keys. Now we set $T = T'$.
5. We define the weight $W(C)$ of the cycle C to be the sum of its edge weights, with constant coefficient $W(C)_0$ and parametric coefficient $W(C)_1$. We store $k(b)$, the maximum vertex key, as an eigenvalue of

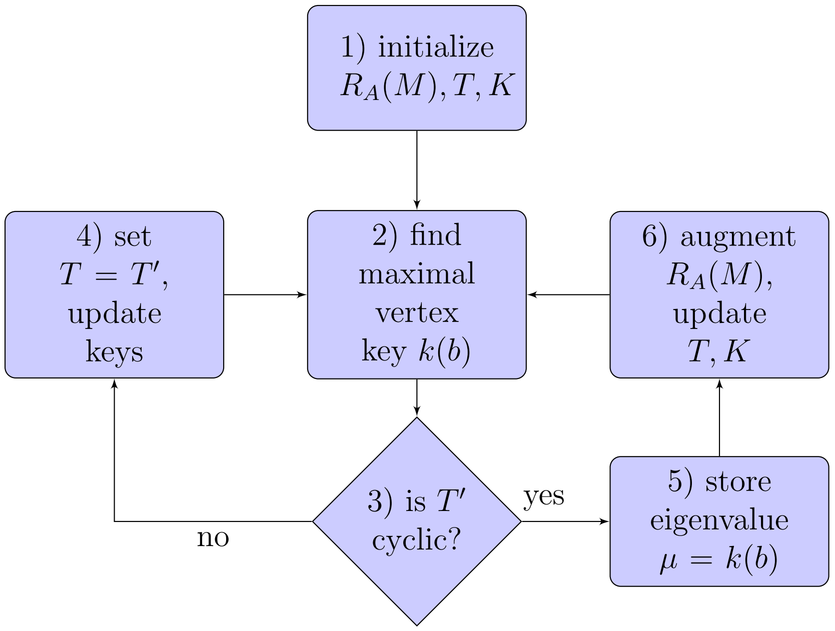

1) initialize $R_A(M), T, K$
2) find maximal vertex key $k(b)$
3) is T' cyclic?
4) set $T = T'$, update keys
5) store eigenvalue $\mu = k(b)$
6) augment $R_A(M)$, update T, K
yes
no

Figure 3: Flow chart for the procedure of max-plus matrix eigenvalues algorithm.

A with multiplicity $-W(C)_1$. If the sum of the multiplicities of the stored eigenvalues is equal to n then we terminate the algorithm.

6. We augment the current maximal matching M_k according to the cycle C to obtain $M_{k+1} = M_k \triangle C$. The graph $R_A(M_{k+1})$ can be obtained directly from $R_A(M_k)$ by reversing the direction and misusing the weight of any edge that belongs to the cycle C.
   Next we construct a new maximally weighted tree T in $R_A(M_{k+1}) \cup r$ rooted at r. From the previous tree we remove the first edge of the directed path from the pivot vertex b to a (where vertex a is the start of the pivot edge e) and replace it with the maximum pivot edge e. Then any edges in this set which are reversed and minused as we update the residual, we also reverse and minus to produce the new tree T.
   Next we update all of the vertex keys.

Example Consider the matrix of Example 1. Figure 4 shows the different graphs produced during the algorithm.

- Subfigure (a). Step 1. We construct the initial residual graph $R_A(M_1) \cup r$ and tree T, then compute the initial vertex keys. For example the vertex $u(1)$ has two edges into it, the edge $e(r)_{r,1}$ has key $k(e(r)_{r,1}) = -\infty$ and the edge $e(1)_{11}$ has key $k(e(1)_{11}) = 2/1 = 2$, so that $k\big(u(1)\big) = \max\{2, -\infty\} = 2$.

- (a)$\mapsto$(b). Steps 2,3,4. The maximal key is $k\big(u(3)\big) = 3$, the pivot edge is the parametric edge $e(1)_{33}$. We update T according to this pivot and do not get any cycles, so we update the vertex keys.

- (b)$\mapsto$(c). Steps 2,3,4. The maximal key is $k\big(u(1)\big) = 2$, the pivot edge is the parametric edge $e(1)_{11}$. We update T according to this pivot and do not get any cycles, so we update the vertex keys .

- (c)$\mapsto$(d). Steps 2,3,5,6. The maximal key is $k\big(u(2)\big) = 2$, the pivot edge is the parametric edge $e(1)_{22}$. We update T according to this pivot and get a cycle $C = \big(u(2), v(1), u(1), v(2)\big)$. Since $W(C) = 4 - 2x$, we save $\mu = 2$ as an eigenvalue of multiplicity 2. We augment M_1 with respect to C to obtain the new maximal matching M_2. We compute the new maximal tree and the new vertex keys for $R_A(M_2) \cup r$.

- (d)$\mapsto$(e). Steps 2,3,4. The maximal key is $k\big(u(1)\big) = 2$, the pivot edge is the root edge $e(r)_{r1}$. We update T according to this pivot and do not get any cycles, so we update the vertex keys.

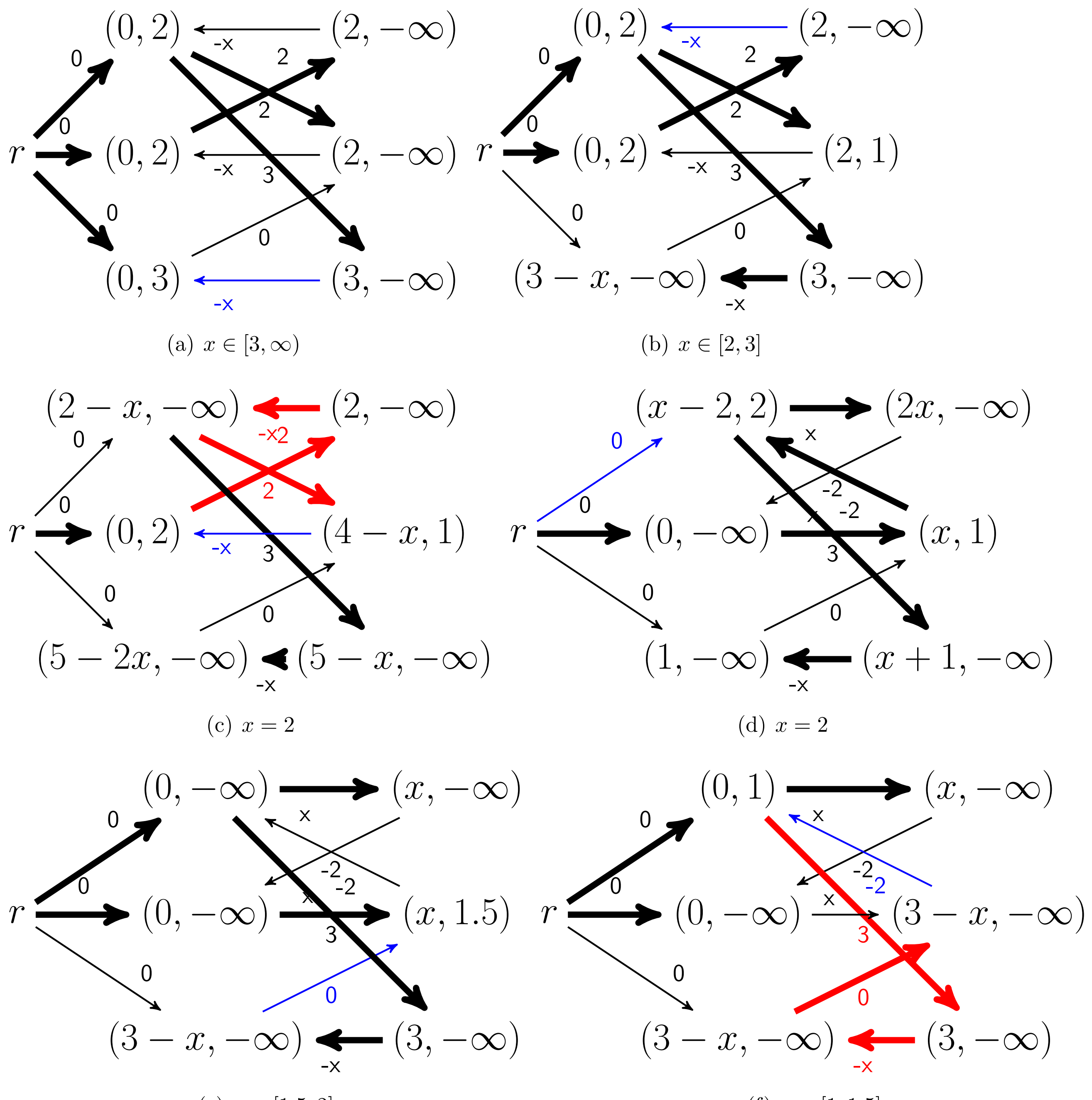
- (e)$\mapsto$(f). Steps 2,3,4. The maximal key is $k\big(v(2)\big) = 1.5$, the pivot edge is the constant edge $e(0)_{32}$. We update T according to this pivot and do not get any cycles, so we update the vertex keys.

- (f). Steps 2,3,5,6. The maximal key is $k\big(u(1)\big) = 1$, the pivot edge is the constant edge $e(0)_{12}$. We update T according to this pivot and get a cycle $C = \big(v(2), u(1), v(3), u(3)\big)$. Since $W(C) = 1 - x$, we save $\mu = 1$ as an eigenvalue of multiplicity 1. The sum of the multiplicities of the stored eigenvalues is $2 + 1 = 3$ so we terminate the algorithm.

- Output. The eigenvalues of A are $\mu = 2$, which is a double eigenvalue and $\mu = 1$, which is a simple eigenvalue.

2.2. Complexity

This algorithm will return all n max-plus eigenvalues of an $n \times n$ max-plus matrix $A \in \mathbb{R}_{\max}^{n\times n}$ with cost $O\big(n\tau_A + n^2\log(n)\big)$, where τ_A is the number of finite entries in A



(a) $x \in [3, \infty)$ (b) $x \in [2, 3]$

(c) $x = 2$ (d) $x = 2$

(e) $x \in [1.5, 2]$ (f) $x \in [1, 1.5]$

Figure 4: Rooted residual graphs $R_A(M) \cup r$ with maximally weighted tree T (bold edges), pivot edges (blue edges), and cycles (red and blue edges). Vertex positions marked with (vertex depth, vertex key) pairs.

The vertex key values are stored in a Fibonacci heap, so that the cost of the operation find maximum is $O(\log(n))$, decrease key is $O\big(\log(n)\big)$ and increase key is $O(1)$. The tree T is stored as an array of parents with doubly linked lists of children, so that the cost of the operation change parent is $O(1)$ and the cost of the operation find all decedents is $O(d)$, where d is the number of decendents. We will treat the cost associated with each component of the flow diagram separately.

Each of the following items derives the complexity for the corresponding stages of the algorithm as described at the start of Subsection 2.1. It may be necessary for the reader to refer back to the algorithm description for some details and notation.

1. The initialization of the residual graph, tree and keys is streightforwards and has cost $O(\tau)$, where τ is the number of non-zeros in the matrix.
2. Find maximum key has cost $O(\log(n))$. Consider $p = \sum_{a\in U\cup V} D_1(a)$, the sum of all of the parametric coefficients of the depths of all of the vertices in $R_A(M) \cup r$. Initially $p = 0$ and at the end of the algorithm $|p| \leq n^2$. Each time the algorithm progresses through stage 4 (the left loop in the flowchart), p decreases by at least one. Each time the algorithm stores a new eigenvalue and moves through stages 5 and 6 (the right loop), p increases by at most n times the multiplicity of the eigenvalue. Thuerefore the total number of times that the algorithm progresses though stage 5 is $O(n^2)$, so that the total cost of finding the maximum keys is $O(n^2 \log(n))$.
3. To determine whether or not T' is cyclic we populate a list with all of the vertices downstream of b in T. If the vertex a appears in this list then T' will be cyclic, otherwise it is not. The cost of this check is $O(\sum_{c\prec b} 1)$, where the sum is taken over all vertices downstream of b in T. Therefore the total cost associated with these operation is $O(\sum_{c\in U\cup V} l(c))$, where $l(c)$ is the total number of times that vertex c is downstream of the pivot vertex b.
   Initially $D(c)_1 = 0$ and at the end of the algorithm $|D(c)_1| \leq n$. Each time that c is downstream of a pivot vertex (the left loop) $D(c)_1$ will decrease by at least one. Each time that the algorithm stores a new eigenvalue (right loop) $D(c)_1$ will increase by at most the multiplicity of the stored eigenvalue. Therefore $l(c)$ is $O(n)$ and the total cost of checking for cyclicity is $O(n^2)$.
4. The cost of switching from T to T' is $O(1)$. We must also update $D(b)$

and $D(c)$ for all vertices $c \in U \cup V$ downstream of b in T. This has cost $O(\sum_{c \prec b} 1)$, the same as checking for cyclicity, so that the total cost of this operation is also $O(n^2)$. The cost of updating the keys after a switching from T to T' is $O\Big(\sum_{c \prec b} \big(\tau_c + \log(n)\big)\Big)$, where the sum is taken over all vertices $c \in U \cup V$ downstream of the vertex with minimal key b and τ_c is the degree of c in $R_A(M)$. This is because for each downstream vertex c we have to check all of the edges incident to c, with cost $O(\tau_c)$, then we may have to increase the vertex keys of c's neighbours, with cost $O(\tau_c)$, and decrease c's own key, with cost $\log(n)$. As we showed in 3 (above), the total number of times that a vertex c is downstream of the pivot vertex is $O(n)$, so that the total cost of updating the keys is $O\big(n\tau_A + n^2 \log(n)\big)$, where we have used the fact that $\sum_{c \in U \cup V} \tau_c = \tau_A$.

5. Storing the new eigenvalue has cost $O(1)$. We do this up to n times so the total cost is $O(n)$.
6. Augmenting $R_A(M)$ has cost $O(n)$, updating T has cost $O(n)$ and updating all of the vertex keys has cost $O(\tau)$. We update in this way at most n times (once for each distinct eigenvalue) so that the total cost associated with these operations is $O(n\tau)$.

3. Matrix polynomial eigenvalues

In this section we detail our extension of Gassner and Klinz's single matrix eigenvalue algorithm to the max-plus matrix polynomial case. Let P be a degree-d, $n \times n$ max-plus matrix polynomial

$$P(x) = \bigoplus_{k=0}^{d} A(k) \otimes x^{\otimes k},$$

where $A(0), \dots, A(d) \in \mathbb{R}_{\max}^{n \times n}$ are the *coefficient matrices* of P. The *max-plus eigenvalues* $\mu_1, \dots, \mu_{nd}$ of P are the non-differentiability points of the *characteristic polynomial*

$$\chi_P(x) = \operatorname{perm}[P(x)] = \max_{\pi \in \Pi_n} \Big(\sum_{i=1}^{n} \big(\max_{k=0}^{d} (A(k)_{i\pi(i)} + kx) \big) \Big).$$

We also include $-\infty$ as an eigenvalue of multiplicity k, whenever the first k coefficients of χ_P are $-\infty$. If the sum of the multiplicities of these eigenvalues is less than nd, then we include $+\infty$ as an eigenvalue with sufficient multiplicity that the sum of the eigenvalues multiplicities is nd.

Like the max-plus matrix eigenproblem, the max-plus eigenvalues of a max-plus matrix polynomial are the switching points of a parametric maximally weighted matching problem.

Let G_P be be the bipartite graph with left vertices $\{u(1), \dots, u(n)\}$, right vertices $\{v(1), \dots, v(n)\}$, and edge sets $E(k) = \{e(k)_{ij} : u(i) \mapsto v(j)$, with weight $w(e(k)_{ij}) = a(k)_{ij} + xk$, for $ij = 1, \dots, n$ such that $a(k)_{ij} \neq -\infty\}$, for $k = 0, 1, \dots, d$. So that the graph described in section 1 corresponds to the special case of the matrix pencil $A \oplus x \otimes I$.

As before a matching or assignment of G_P is a subset of edges $M \subset E(0) \cup \dots \cup E(d)$, such that each U vertex has exactly one edge out of it and each V vertex has exactly one edge into it. Therefore the edges in an assignment M can be uniquely represented as $M = M(\pi, s) = \{(e(s_i)_{i,\pi(i)} : i = 1, \dots, n\}$, for some $s \in \{0, 1, \dots, d\}^n$ and some $\pi \in \Pi_n$, a permutation on $\{1, \dots, n\}$. The weight of an assignment is simply the sum of its edge weights

$$W\big(M(\pi, s)\big) = \sum_{k=1}^{n} w\big(e(s_k)_{i\pi(i)}\big),$$

so that just as in the single matrix eigenproblem we have

$$\chi_P(x) = \max_{\pi \in \Pi_n} \max_{s \in \{0,1,\dots,d\}^n} \sum_{k=1}^{n} w\big(e(s_k)_{i,\pi(i)}\big) = \max_{M} W(M), \tag{4}$$

where the rightmost maximum is taken over all assignments of G_P. The switching values of x where the assignment that attain the maximum in (4) changes are the non-differentiability points of $\chi_P(x)$, which are the finite max-plus eigenvalues $\mu_1, \dots, \mu_{nd}$ of P.

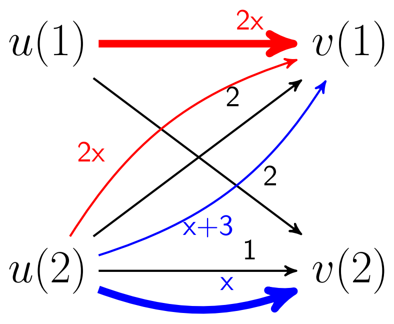
We define the residual bipartite graph $R_P(M)$ to the graph obtained from G_P by reversing the direction of an minusing the weight of any edges that appear in M.

Example Consider the max-plus matrix polynomial

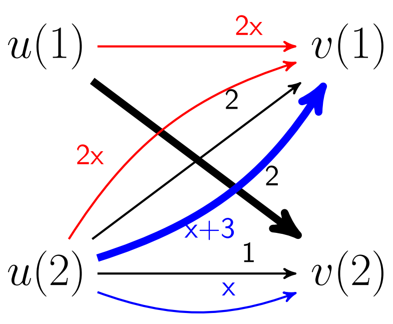
$$P(x) = \begin{bmatrix} 0 & -\infty \\ 0 & -\infty \end{bmatrix} \otimes x^{\otimes 2} \oplus \begin{bmatrix} -\infty & \infty \\ 3 & 0 \end{bmatrix} \otimes x \oplus \begin{bmatrix} -\infty & 2 \\ 2 & 1 \end{bmatrix}.$$

Expanding the characteristic polynomial of P yields

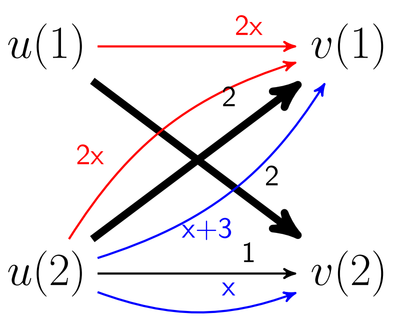
$$\chi_P(x) = x^{\otimes 3} \oplus 2 \otimes x^{\otimes 2} \oplus 5 \otimes x \oplus 4 = \max\{3x, 2x + 2, x + 5, 4\}.$$



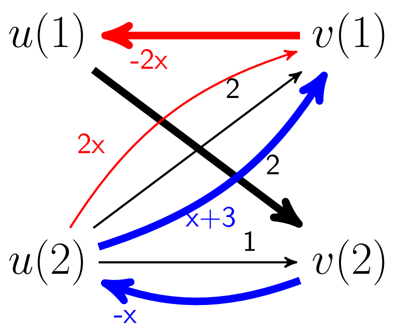
(a) $x \in [2.5, \infty)$,
$M_1 = \{e(2)_{11}, e(2)_{22}\}$.



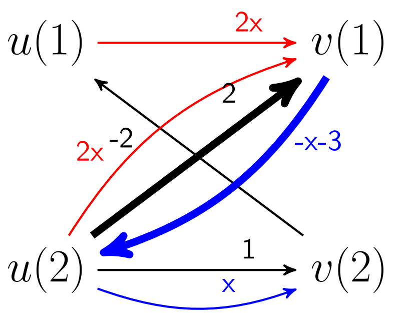
(b) $x \in [-1, 2.5]$,
$M_2 = \{e(0)_{12}, e(1)_{21}\}$.



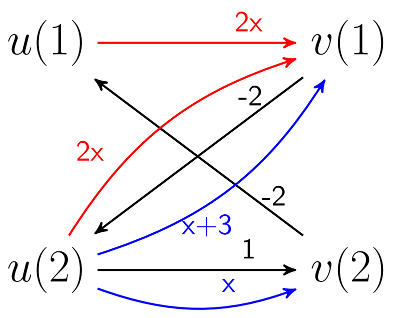
(c) $x \in [-\infty, -1]$,
$M_3 = \{e(0)_{12}, e(0)_{21}\}$.



(d) $R_P(M_1)$,
$C_1 =$
$\{e(2)_{11}, e(1)_{21}, e(2)_{22}, e(0)_{12}\}$.



(e) $R_P(M_2)$,
$C_2 = \{e(1)_{12}, e(0)_{12}\}$.



(f) $R_P(M_3)$

Figure 5: a,b,c) Bipartite graph G_A, with maximally weighted matchings (bold edges). d,e,f) Residual graphs with augmenting cycles (bold edges).

The roots of χ_P and therefore the finite eigenvalues of P are given by $\mu_1 = 2.5$, which is a double root, and $\mu_2 = -1$, which is a simple root. The sum of the multiplicity of these roots is $2 + 1 = 3$, so we include $\mu_3 + \infty$. as an eigenvalue of multiplicity 1. The three different assignments M_i $i = 1, 2, 3$ that attain the maximum in (4) are displayed in Figure 5 subfigures a,b,c. Note that the intervals in which a particular matching is maximally weighted are always bounded by pairs of max-plus eigenvalues of P. Figure 5 subfigures d,e,f displays the residual graphs $R_P(M_i), i = 1, 2, 3$ with augmenting cycles highlighted. Note that $M_2 = M_1 \triangle C_1$ and that $M_3 = M_2 \triangle C_2$.

Like the single matrix eigenvalue algorithm, our matrix polynomial algorithm works by maintaining a maximally weighted spanning tree through a residual bipartite graph and finding eigenvalues by detecting weight zero cycles, at which point the maximally weighted trees cease to exist and the maximally weighted matching must be updated.

There are two main differences between our max-plus matrix polynomial algorithm and Gassner and Kilnz's single matrix eigenvalue algorithm. The first difference is that computing the initial maximally weighted matching and the initial maximally weighted tree is a little more complicated. The second difference is that we begin by computing the max-plus roots of each entry of the matrix polynomial (recall that each entry is itself a degree-d max-plus scalar polynomial). This enables us to work with a residual bipartite graph with a single set of edges (rather than d sets of edges). Each time the parameter x passes through a root of p_{ij}, we update the edge weight of the edge $e : i \mapsto j$ accordingly. Essentially the individual entry roots form a special set of keys, which we can treat separately to speeds things up.

3.1. The algorithm

This algorithm will return all nd max-plus eigenvalues of a degree-d, $n \times n$ max-plus matrix polynomial P with cost $O\big(nd\tau + n^2 d\log(n) + \tau d\log(d)\big)$, where τ is the number of finite entries in P. Figure 6 is a flow chart which summarizes the algorithm's procedure.

1. For $ij = 1, \dots, n$ we compute the finite max-plus roots $s_{ij}(1), \dots, s_{ij}(t)$ and their multiplicities $m_{ij}(1), \dots, m_{ij}(t)$ of the max-plus polynomial

$$p(x)_{ij} = a(o)_{ij} \oplus a(1)_{ij} \otimes x \oplus \dots \oplus a(d)_{ij} \otimes x^{\otimes d}. \tag{5}$$

For each root $s_{ij}(k)$ we also store $w_{ij}(k)$, the monomial which attains the maximum in (5) for $x \in [s_{ij}(k-1), s_{ij}(k)]$. We combine all of the roots from each entry into one list S, which we sort according to the value of the root in decreasing order. Note that we do not store any infinite roots.
We compute the optimal assignment for large positive x. We define an order $\preccurlyeq$ on affine functions of the form $a + bx$, with $a, b \in \mathbb{R}_{\max}$, by $a + bx \preccurlyeq c + dx$ if either $a = -\infty$, $b < d$ and $c \neq -\infty$ or $b = d$ and $a \leq c$. Thus $a + bx \preccurlyeq c + dx$ if and only if $a + bx \leq c + dx$, for large positive x.

Define $Q(x)$ to be the max-plus matrix of monomials whose entires $q(x)_{ij}$ are the greatest (with respect to the ordering $\preccurlyeq$) monomials in the polynomial entries $p(x)_{ij}$. Thus $q(x)_{ij}$ is the highest degree monomial with a finite coefficient in $p(x)_{ij}$.
We compute the optimal assignment M_1 of $Q(x)$, using the order $\preccurlyeq$ to compare the weights of different assignments. This is carried out by the Hungarian algorithm, which only needs to add, subtract and compare matrix entries. We substitute the usual operations for the analogous operations on functions of the form $a+bx$ and also use our new ordering $\preccurlyeq$.
If the parametric component of the optimal assignments weight $W(M_1)_1$ is less than nd then we store $\mu = +\infty$ as an eigenvalue with multiplicity $nd - W(M_1)_1$. If there is no finite weight assignment of Q, then we terminate the algorithm and report that P is degenerate.
We construct the initial residual bipartite graph $R_P(M_1)$ with left vertices $\{u(1), \dots, u(n)\}$, right vertices $\{v(1), \dots, v(n)\}$ and edge set $E = \{e_{ij} : u(i) \mapsto v(j)$, with weight $w(e_{ij}) = q(x)_{ij}$, for $ij = 1, \dots, n$ such that $q(x)_{ij} \neq -\infty\}$. We reverse the direction of and minus the weight of any edges that appear in the optimal assignment M_1.
We adjoin a root vertex r, with constant edge set $E(r) = \{e(r)_{ri} : r \mapsto u(i)$, with weight $w(e(r)_{ri}) = 0$, for $i = 1, \dots, n\}$. We compute the maximally weighted spanning tree T through $R_P(M_1) \cup r$ rooted at r, using Edmond's algorithm with the ordering $\preccurlyeq$. As with computing the optimal assignment , this is achieved by substituting the standard numerical operations, for operations on affine functions of the form $a + bx$.
We define the depth $D(a)$ of a vertex $a \in U \cup V$, and the edge and vertex keys exactly as in the single matrix algorithm. We compute all of the vertex keys.

2. We take the maximum vertex key, $k(b)$ with origin e. We call b the *pivot vertex* and $e : a \mapsto b$ the *pivot edge*. If this maximum is not unique, then it does not matter which maximal vertex key we choose.
We also take the maximum entry $s_{ij}(k)$ from the list S. If this maximum is not unique, then it does not matter which maximal root we choose.
If there are no finite keys or roots then we store minus infinity as an eigenvalue with multiplicity equal to n minus the sum of the multiplicities of the finite eigenvalues stored so far and terminate the algorithm.

3. We pick the maximum out of $k(b)$ and $s_{ij}(k)$. If $s_{ij}(k)$ is the maximum then we remove it from the list S. If this maximum is not unique then it does not matter which of $k(b)$ and $s_{ij}(k)$ we choose.
4. This step is exactly the same as in the single matrix eigenvalue algorithm. If b is the pivot vertex and $e : a \mapsto b$ is the pivot edge, then T' is cyclic if and only if a is downstream of b in T.
5. This step is exactly the same as in the single matrix eigenvalue algorithm, except that there is only one set of edges that need checking. We update the key of any edge incident to a vertex downstream of the pivot vertex, then update the vertex keys accordingly.
6. This step is exactly the same as in the single matrix eigenvalue algorithm. We store the eigenvalue $\mu = k(b)$, with multiplicity equal to the parametric coefficient of the weight of the cycle.
7. This step is exactly the same as in the single matrix eigenvalue algorithm, except that there is only one set of edges that need checking when we update K.
8. We check weather the edge e_{ij} corresponding to the maximal matrix entry root is currently a forwards edge $e_{ij} : u(i) \mapsto v(j)$ or if it has been reversed to a backwards edge $e_{ij} : v(j) \mapsto u(i)$.
9. We update $R_P(M)$ by setting the weight of the edge e_{ij} to $w_{ij}(k)$. As in step 5 we must now update the keys of all edges incident to vertices downstream of $v(j)$ in T, then update the vertex keys accordingly.
10. We store the eigenvalues $\mu = s_{ij}(k)$, with multiplicity $m_{ij}(k)$.
11. We update $R_P(M)$ by setting the weight of the backwards edge e_{ij} to $-w_{ij}(k)$. As in step 5 we must now update the keys of all edges incident to vertices downstream of $u(i)$ in T, then update the vertex keys accordingly.

Example Consider the max-plus matrix polynomial of Example 3. Figure 7 shows the different graphs produced during the algorithm.

- Subfigure (a). Step 1. We compute the finite roots of all of the matrix polynomial entries. The $(1,1)$ entry $p(x)_{1,1} = x^{\otimes 2}$ has no finite roots. The $(2,1)$ entry $p(x)_{12} = x^{\otimes 2} \oplus 3 \otimes x \oplus x = \max\{2x, x+3, 2\}$ has two finite roots, $s_{12}(1) = 3$, with $m_{12}(1) = 1$ and $w_{12}(1) = x + 3$, and $s_{12}(2) = -1$, with $m_{12}(2) = 1$ and $w_{12}(1) = 2$. The $(1,2)$ entry $p(x)_{21} = 2$ has no finite roots. The $(2,2)$ entry $p(x)_{22} = x \oplus 1 = \max\{x, 1\}$ has

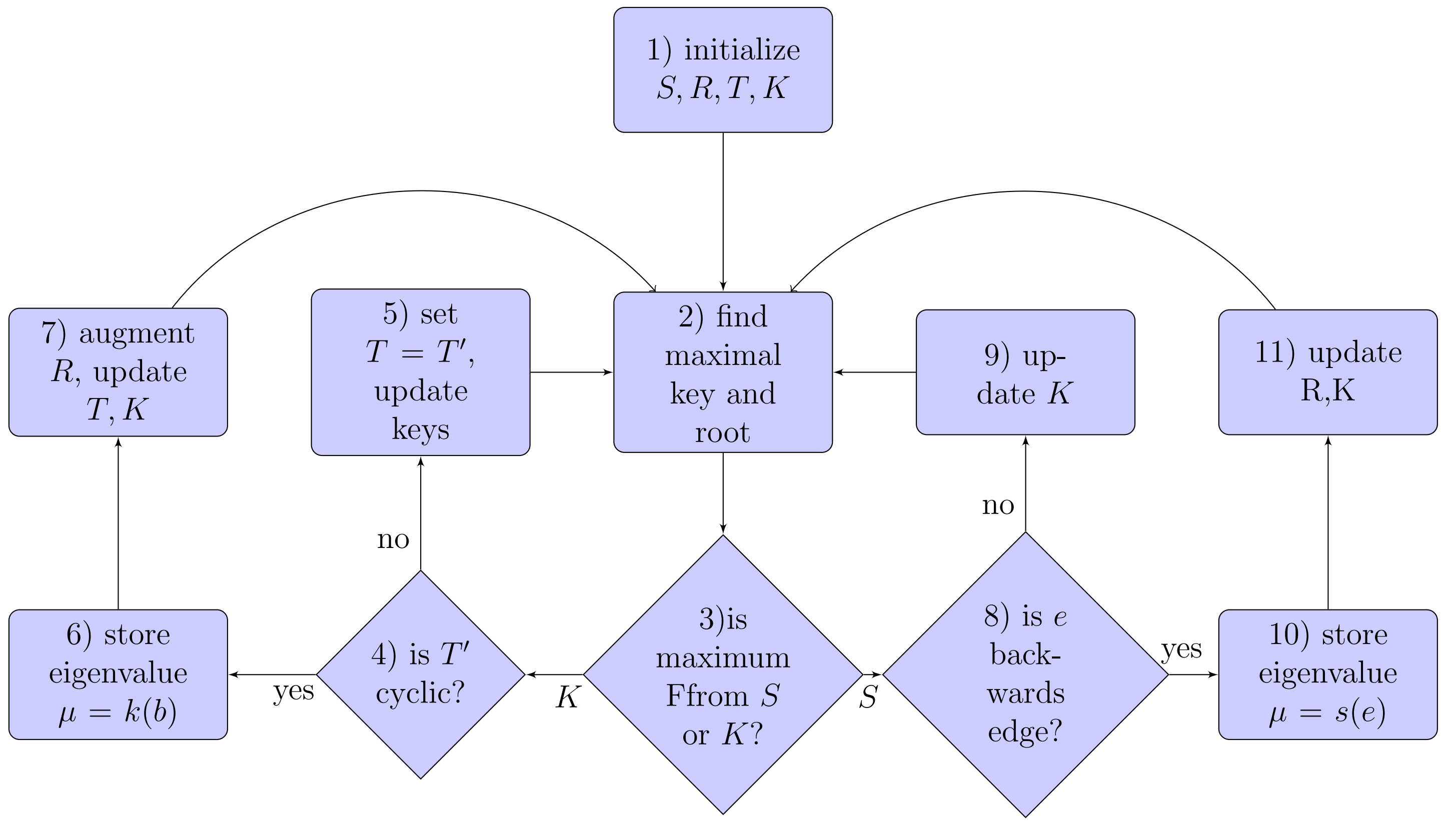

1) initialize S, R, T, K
2) find maximal key and root
3)is maximum Ffrom S or K?
4) is T' cyclic?
5) set $T = T'$, update keys
6) store eigenvalue $\mu = k(b)$
7) augment R, update T, K
8) is e back-wards edge?
9) up-date K
10) store eigenvalue $\mu = s(e)$
11) update R,K
K
S
yes
no
no
yes

Figure 6: Flow chart for the procedure of max-plus matrix polynomial eigenvalues algorithm.

one finite root, $s_{22}(1) = 1$, with $m_{22}(1) = 1$ and $w_{22}(1) = 1$. The ordered list of roots is given by $S = \big(s_{12}(1) = 2, s_{22}(1) = 1, s_{12}(2) = -1\big)$.

The matrix $Q(x)$ is given by

$$Q(x) = \begin{bmatrix} 2x & 2 \\ 2x & x \end{bmatrix}.$$

The maximally weighted matching M_1 of G with respect to the ordering $\preccurlyeq$ is therefore $M_1 = \{e(2)_{11}, e(1)_{22}\}$, which has weight $3x$, so we include $\mu = +\infty$ as an eigenvalue with multiplicity $4 - 3 = 1$.

- (a)$\mapsto$(b). Steps 2,3,8,9. The maximal key is $k\big(u(2)\big) = 2$. The maximal root is $s_{12}(1) = 3$, which corresponds to the forwards edge $\big(u(2), v(1)\big)$. Since the root is the larger it corresponds to a forwards edge we set the weight of this edge to $w_{12}(1) = x + 3$ then update K.

- (b)$\mapsto$(c). Steps 2,3,4,5. The maximal key is $k\big(u(1)\big) = 3$, with pivot edge $\big(u(1), v(1)\big)$. The maximal root is $s_{22}(1) = 1$. Since the key is larger we check T'. Since there are no cycles in T' we set $T = T'$ and update K.

- (c)$\mapsto$(d). Steps 2,3,4,6,7. The maximal key is $k\big(u(2)\big) = 2.5$, with pivot edge $\big(u(2), v(2)\big)$. The maximal root is $s_{22}(1) = 1$. Since the key is larger we check T'. We check T' and find a cycle $C = \big(u(1), v(1), u(2), v(2)\big)$. We store $\mu = 2.5$ as an eigenvalue of multiplicity 2. We augment G with respect to C and update T and K.

- (d)$\mapsto$(e). Steps 2,3,4,5. The maximal key is $k\big(u(1)\big) = 2$, with pivot edge $\big(u(1), r\big)$. The maximal root is $s_{22}(1) = 1$. Since the key is larger we check T'. Since there are no cycles in T' we set $T = T'$ and update K.

- (e)$\mapsto$(f). Steps 2,3,8,9. There are no finite keys. The maximal root is $s_{22}(1) = 1$, which corresponds to the forwards edge $\big(u(2), v(2)\big)$. Since the root corresponds to a forwards edge we set the weight of this edge to $w_{22}(1) = 1$ then update K.

- (f). Steps 2,3,8,10,11. There are no finite keys. The maximal root is $s_{12}(2) = -1$. This root corresponds to the backwards edge $\big(u(2), v(1)\big)$.

Since the root corresponds to a backwards edge we store $\mu = -1$ as an eigenvalue with multiplicity one. The sum of the multiplicities of the strored eigenvalues is now $1 + 2 + 1 = 4$ so we terminate the algorithm.

- Output. The eigenvalues of P are therefore $\mu_1 = +\infty$, which is a simple eigenvalue, $\mu_2 = 2$, which is a double eigenvalue and $\mu_3 = 0$, which is a simple eigenvalue.

3.2. Complexity

This algorithm will return all nd max-plus eigenvalues of a degree-d, $n \times n$ max-plus matrix polynomial P with cost $O\big(nd\tau_P + n^2 d\log(n) + \tau_P d\log(d)\big)$, where τ_P is the number of entries in P that are not identically equal to minus infinity. So that for $d \leq \exp(n)$ and $\tau_P \geq n\log(n)$ we have

$$nd\tau_P + n^2 d\log(n) + \tau d\log(d) = O(nd\tau_P).$$

As before the vertex key values are stored in a Fibonacci heap and the tree T is stored as an array of parents with doubly linked lists of children. We will treat the cost associated with each component of the flow diagram separately.

The advantage to treating the matrix entry roots separately from the other keys is that when we update vertex keys, we only need to check the edge keys for a single set of edges, rather than d different sets. This improves the complexity by a factor of d.

1. The roots of each matrix polynomial entry $p_{ij}(x)$ can be computed by applying the Graham scan algorithm to the newton polygon of p_{ij}, with cost $O(d)$. The total cost for computing all of the roots is therefore $O(d\tau_P)$. The roots for each entry are already sorted, so the list of all roots can be sorted using merge sort with cost $O\big(\tau_P d\log(d)\big)$.
   Computing the initial optimal assignment with the Hungarian algorithm has cost $O\big(n\tau_P + n\log(n)\big))$.
   Computing the initial maximally weighted tree with Edmond's algorithm has cost $O\big(\tau_P + n\log(n)\big)$.
   Computing the initial keys has cost $O(\tau)$.
2. Find maximum key has cost $O(\log(n))$. Find maximum root has cost $O(1)$. Consider $p = \sum_{a\in U\cup V} D_1(a)$, the sum of all of the parametric coefficients of the depths of all of the vertices in $R_P(M)$. Initially $|p| \leq n^2 d$ and at the end of the algorithm $|p| \leq n^2 d$. Each time the

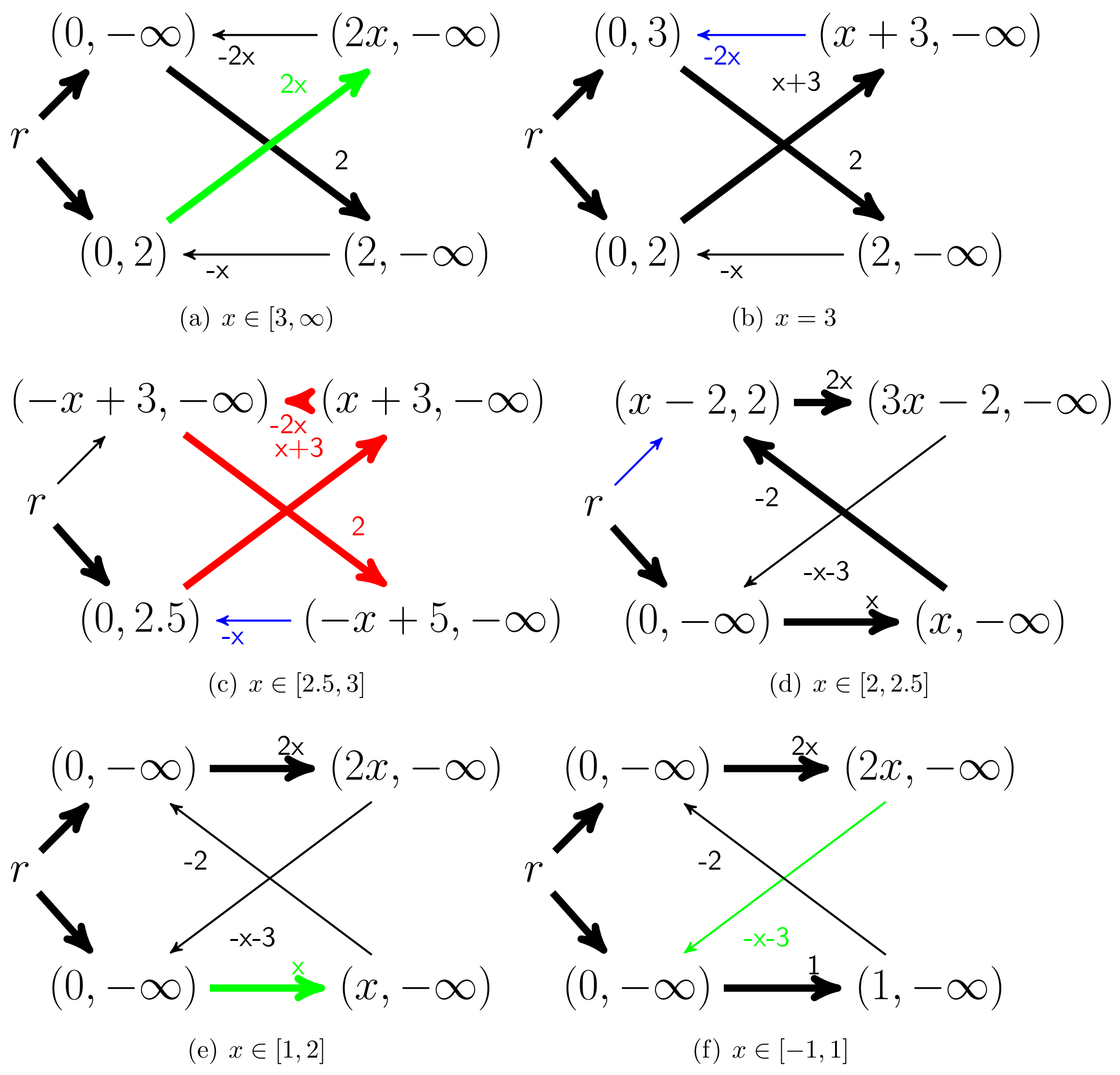


Figure 7: Residual graph R with maximally weighted tree (bold edges), pivot edges (blue edges), cycles (red and blue edges) and updated edges (green). Vertex positions marked with (vertex depth, vertex key) pairs.

algorithm progresses through stage 5 or 9 (the left or right inner loop in the flowchart), p decreases by at least one. Each time the algorithm stores a new eigenvalue and moves through stages 6 or 10 (the left or right outer loop), p increases by at most n times the multiplicity of the stored eigenvalue. Since the sum of the stored eigenvalues multiplicities comes to nd, the total number of times that the algorithm progresses though stage 2 is $O(n^2d)$, so that the total cost of fining the maximum keys is $O(n^2d\log(n))$.

3. Choosing the maximum of the maximum key and the maximum root has cost $O(1)$, as explained in 1 (above) we do this $O(n^2d)$ times, so the total cost is $O(n^2d)$.
4. To determine wheather or not T' is cyclic we populate a list with all of the vertices downstream of b in T. If the vertex a appears in this list then T' will be cyclic, otherwise it is not. The cost of this check is $O(\sum_{c\prec b}1)$, where the sum is taken over all vertices $c\in U\cup B$ downstream of b in T. Therefore the total cost associated with this operation is $O(\sum_{c\in U\cup V}l(c))$, where $l(c)$ is the total number of times that vertex c is downstream of the pivot vertex b.
   Initially $|D(c)_1|\leq dn$ and at the end of the algorithm $|D(c)_1|\leq dn$. Each time that c is downstream of a pivot vertex (the left loop) $D(c)_1$ will decrease by at least one. Each time that the algorithm stores a new eigenvalue (right loop) $D(c)_1$ will increase by at most the multiplicity of the stored eigenvalue. Therefore $l(c)$ is $O(nd)$ and the total cost of checking for cyclicity is $O(dn^2)$.
5. The cost of switching from T to T' is $O(1)$, we must also update $D(b)$ and $D(c)$ for all vertices $c\in U\cup V$ downstream of b in T. This has cost $O(\sum_{c\prec b}1)$ the same as checking for cyclicity, so that the total cost of this operation is also $O(n^2d)$. The cost of updating the keys after switching from T to T' is $O\Big(\sum_{c\prec b}\big(\tau_c+\log(n)\big)\Big)$, where the sum is taken over all vertices c, downstream of the vertex with minimal key b and τ_c is the degree of c in R. This is because we have to check all of the edges incident to each downstream vertex $c\in U\cup v$, with cost $O(\tau_c)$, then we may have to increase the vertex keys of c's neighbours, with cost $O(\tau_c)$, and decrease c's own key, with cost $\log(n)$.
   As we showed in 4 (above), the total number of times that a vertex c is downstream of the pivot vertex is $O(dn)$, so that the total cost of updating the keys is $O\big(nd\tau_P+n^2d\log(n)\big)$.

6. Storing the new eigenvalue has cost $O(1)$. We do this up to nd times so the total cost is $O(nd)$.
7. Augmenting $R_P(M)$ has cost $O(n)$, updating T has cost $O(n)$ and updating all of the vertex keys has cost $O(\tau)$. We update in this way at most nd times (once for each distinct eigenvalue) so that the total cost associated with these operations is $O(nd\tau)$.
8. Checking whether the edge e is backwards has cost $O(1)$, as explained in 1 we do this $O(nd)$ times so the total cost associated with these operations is $O(nd)$.
9. As in 5, we must update the keys for all the edges incident to vertices downstream of the updated edge that the total cost associated with these operations is $O\big(nd\tau + n^2 d\log(n)\big)$.
10. Storing the new eigenvalue has cost $O(1)$. We do this up to nd times so the total cost is $O(nd)$.
11. Updating $R_P(M)$ has cost $O(1)$, updating T has cost $O(n)$ and updating all of the vertex keys has cost $O(\tau)$. We update in this way at most nd times (once for each distinct eigenvalue) so that the goal cost associated with these operations is $O(nd\tau)$.

[1] Elisabeth Gassner and Bettina Klinz. A fast parametric assignment algorithm with applications in max-algebra. *Networks*, 55(2):61–77, 2010.

[2] Stéphane Gaubert and Meisam Sharify. Tropical scaling of polynomial matrices. In *Positive systems*, volume 389 of *Lecture Notes in Control and Information Sciences*, pages 291–303. ,, 2009.

[3] Sven Hammarling, Christopher J. Munro, and Françoise Tisseur. An algorithm for the complete solution of quadratic eigenvalue problems. 39(3):18:1–18:19, April 2013.

[4] D. A. Bini, V. Noferini, and M. Sharify. Locating the eigenvalues of matrix polynomials. 34(4):1708–1727, 2013.

[5] James Hook and Françoise Tisseur. Max-plus eigenvalues and singular values: a useful tool in numerical linear algebra, 2015. In preparation.

[6] James Hook. Max-plus singular values. 486:419–442, 2015.

[7] M. Akian, Ravindra Bapat, and S. Gaubert. Perturbation of eigenvalues of matrix pencils and optimal assignment problem. *C. R. Acad. Sci. Paris, Série I*, (339):103–108, 2004.

[8] R.L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[9] Neal E. Young, Robert E. Tarjan, and James B. Orlin. Faster parametric shortest path and minimum balance algorithms. *Networks*, 1(2):205–221, 1991.

[10] Rainer E. Burkard and Peter Butkovi. Finding all essential terms of a characteristic maxpolynomial. *Discrete Applied Mathematics*, 130(3):367 – 380, 2003.

[11] Meisam Sharify. *Scaling Algorithms and Tropical Methods in Numerical Matrix Analysis: Application to the Optimal Assignment Problem and to the Accurate Computation of Eigenvalues.* PhD thesis, Ecole Polytechnique, Palaiseau, France, September 2011.