Parallel Computation

Matt Williamson[1]

[1] Lane Department of Computer Science and Electrical Engineering
West Virginia University

Algorithms, Models, Classes NC and RNC

Outline

Outline

Outline

Outline

Parallel Computers

Setup

We think of a parallel computer with a large number of independent processors, where each processor can execute its own program, and can communicate with other processors instantaneously and synchronously through a large shared memory. In other words, all processors execute their first instruction in unison, then they exchange information, then they execute the second instruction, and so on. There are other kinds of multiprocessors, but this is the easiest one to think about and for writing algorithms.

Goal

When designing algorithms for parallel computers, we want to minimize the time between the beginning and the end of the concurrent computation. Specifically, we want our parallel algorithms to be faster than our sequential ones. Naturally, our algorithms should require a realistic number of processors.

Parallel Computers

Setup

We think of a parallel computer with a large number of independent processors, where each processor can execute its own program, and can communicate with other processors instantaneously and synchronously through a large shared memory. In other words, all processors execute their first instruction in unison, then they exchange information, then they execute the second instruction, and so on. There are other kinds of multiprocessors, but this is the easiest one to think about and for writing algorithms.

Goal

When designing algorithms for parallel computers, we want to minimize the time between the beginning and the end of the concurrent computation. Specifically, we want our parallel algorithms to be faster than our sequential ones. Naturally, our algorithms should require a realistic number of processors.

Parallel Computers

Setup

We think of a parallel computer with a large number of independent processors, where each processor can execute its own program, and can communicate with other processors instantaneously and synchronously through a large shared memory. In other words, all processors execute their first instruction in unison, then they exchange information, then they execute the second instruction, and so on. There are other kinds of multiprocessors, but this is the easiest one to think about and for writing algorithms.

Goal

When designing algorithms for parallel computers, we want to minimize the time between the beginning and the end of the concurrent computation. Specifically, we want our parallel algorithms to be faster than our sequential ones. Naturally, our algorithms should require a realistic number of processors.

Parallel Computers

Setup

We think of a parallel computer with a large number of independent processors, where each processor can execute its own program, and can communicate with other processors instantaneously and synchronously through a large shared memory. In other words, all processors execute their first instruction in unison, then they exchange information, then they execute the second instruction, and so on. There are other kinds of multiprocessors, but this is the easiest one to think about and for writing algorithms.

Goal

When designing algorithms for parallel computers, we want to minimize the time between the beginning and the end of the concurrent computation. Specifically, we want our parallel algorithms to be faster than our sequential ones. Naturally, our algorithms should require a realistic number of processors.

Parallel Computers

Setup

We think of a parallel computer with a large number of independent processors, where each processor can execute its own program, and can communicate with other processors instantaneously and synchronously through a large shared memory. In other words, all processors execute their first instruction in unison, then they exchange information, then they execute the second instruction, and so on. There are other kinds of multiprocessors, but this is the easiest one to think about and for writing algorithms.

Goal

When designing algorithms for parallel computers, we want to minimize the time between the beginning and the end of the concurrent computation. Specifically, we want our parallel algorithms to be faster than our sequential ones. Naturally, our algorithms should require a realistic number of processors.

Parallel Computers

Setup

We think of a parallel computer with a large number of independent processors, where each processor can execute its own program, and can communicate with other processors instantaneously and synchronously through a large shared memory. In other words, all processors execute their first instruction in unison, then they exchange information, then they execute the second instruction, and so on. There are other kinds of multiprocessors, but this is the easiest one to think about and for writing algorithms.

Goal

When designing algorithms for parallel computers, we want to minimize the time between the beginning and the end of the concurrent computation. Specifically, we want our parallel algorithms to be faster than our sequential ones. Naturally, our algorithms should require a realistic number of processors.

Outline

Matrix Multiplication

Problem

Suppose that we are given two n x n matrices A and B, and we wish to compute their product $C = A \cdot B$. In other words, we want to compute all n^2 sums of the form

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, \; i, j = 1, \ldots, n.$$

Example

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} = \begin{pmatrix} 90 & 100 & 110 & 120 \\ 202 & 228 & 254 & 280 \\ 314 & 356 & 398 & 440 \\ 426 & 484 & 542 & 600 \end{pmatrix}$$

Note

The standard approach would take $O(n^3)$ arithmetic operations.

Matrix Multiplication

Problem

Suppose that we are given two n x n matrices A and B, and we wish to compute their product $C = A \cdot B$. In other words, we want to compute all n^2 sums of the form

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, i, j = 1, \ldots, n.$$

Example

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} = \begin{pmatrix} 90 & 100 & 110 & 120 \\ 202 & 228 & 254 & 280 \\ 314 & 356 & 398 & 440 \\ 426 & 484 & 542 & 600 \end{pmatrix}$$

Note

The standard approach would take $O(n^3)$ arithmetic operations.

Matrix Multiplication

Problem

Suppose that we are given two n x n matrices A and B, and we wish to compute their product $C = A \cdot B$. In other words, we want to compute all n^2 sums of the form

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj},\ i, j = 1, \ldots, n.$$

Example

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} = \begin{pmatrix} 90 & 100 & 110 & 120 \\ 202 & 228 & 254 & 280 \\ 314 & 356 & 398 & 440 \\ 426 & 484 & 542 & 600 \end{pmatrix}$$

Note

The standard approach would take $O(n^3)$ arithmetic operations.

Matrix Multiplication

Problem

Suppose that we are given two n x n matrices A and B, and we wish to compute their product $C = A \cdot B$. In other words, we want to compute all n^2 sums of the form

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, i, j = 1, ..., n.$$

Example

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} = \begin{pmatrix} 90 & 100 & 110 & 120 \\ 202 & 228 & 254 & 280 \\ 314 & 356 & 398 & 440 \\ 426 & 484 & 542 & 600 \end{pmatrix}$$

Note

The standard approach would take $O(n^3)$ arithmetic operations.

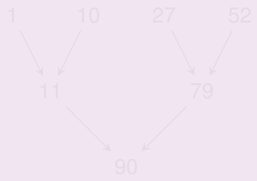
Matrix Multiplication (Contd.)

Maximize Parallelism

One way is to compute each of the n^3 operations in a separate processor, requiring n^3 processors. Then, for each C_{ij} (a total of n^2 processors), we can get its sum by collecting the n products that correspond to it in $n - 1$ additional steps. This gives us a total of n arithmetic operations using n^3 processors. Can we do better?
Yes! We can use binary trees for our addition. With each step, we cut the number of additions by half, giving us $\log n$ steps using n^3 processors.

Example

We want to add 1, 10, 27, and 52 to get $C_{1,1} = 90$.



Matrix Multiplication (Contd.)

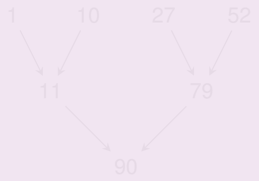
Maximize Parallelism

One way is to compute each of the n^3 operations in a separate processor, requiring n^3 processors. Then, for each C_{ij} (a total of n^2 processors), we can get its sum by collecting the n products that correspond to it in $n-1$ additional steps. This gives us a total of n arithmetic operations using n^3 processors. Can we do better?

Yes! We can use binary trees for our addition. With each step, we cut the number of additions by half, giving us $\log n$ steps using n^3 processors.

Example

We want to add 1, 10, 27, and 52 to get $C_{1,1} = 90$.



Matrix Multiplication (Contd.)

Maximize Parallelism

One way is to compute each of the n^3 operations in a separate processor, requiring n^3 processors. Then, for each C_{ij} (a total of n^2 processors), we can get its sum by collecting the n products that correspond to it in $n-1$ additional steps. This gives us a total of n arithmetic operations using n^3 processors. Can we do better?
Yes! We can use binary trees for our addition. With each step, we cut the number of additions by half, giving us $\log n$ steps using n^3 processors.

Example

We want to add 1, 10, 27, and 52 to get $C_{1,1} = 90$.

$$\begin{tikzcd}
1 \arrow[dr] & & 10 \arrow[dl] & & 27 \arrow[dr] & & 52 \arrow[dl] \\
& 11 \arrow[drr] & & & & 79 \arrow[dll] & \\
& & & 90 & & &
\end{tikzcd}$$

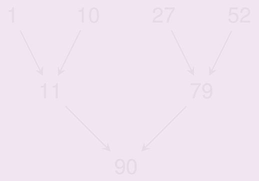
Matrix Multiplication (Contd.)

Maximize Parallelism

One way is to compute each of the n^3 operations in a separate processor, requiring n^3 processors. Then, for each C_{ij} (a total of n^2 processors), we can get its sum by collecting the n products that correspond to it in $n - 1$ additional steps. This gives us a total of n arithmetic operations using n^3 processors. Can we do better?
Yes! We can use binary trees for our addition. With each step, we cut the number of additions by half, giving us $\log n$ steps using n^3 processors.

Example

We want to add 1, 10, 27, and 52 to get $C_{1,1} = 90$.



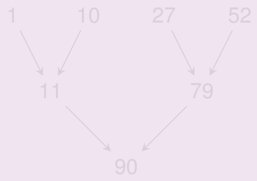
Matrix Multiplication (Contd.)

Maximize Parallelism

One way is to compute each of the n^3 operations in a separate processor, requiring n^3 processors. Then, for each C_{ij} (a total of n^2 processors), we can get its sum by collecting the n products that correspond to it in $n - 1$ additional steps. This gives us a total of n arithmetic operations using n^3 processors. Can we do better?
Yes! We can use binary trees for our addition. With each step, we cut the number of additions by half, giving us $\log n$ steps using n^3 processors.

Example

We want to add 1, 10, 27, and 52 to get $C_{1,1} = 90$.


$$\begin{tikzcd}
1 \arrow[dr] & & 10 \arrow[dl] & & 27 \arrow[dr] & & 52 \arrow[dl] \\
& 11 \arrow[drr] & & & & 79 \arrow[dll] & \\
& & & 90 & & &
\end{tikzcd}$$

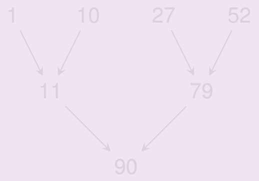
Matrix Multiplication (Contd.)

Maximize Parallelism

One way is to compute each of the n^3 operations in a separate processor, requiring n^3 processors. Then, for each C_{ij} (a total of n^2 processors), we can get its sum by collecting the n products that correspond to it in $n - 1$ additional steps. This gives us a total of n arithmetic operations using n^3 processors. Can we do better?
Yes! We can use binary trees for our addition. With each step, we cut the number of additions by half, giving us $\log n$ steps using n^3 processors.

Example

We want to add 1, 10, 27, and 52 to get $C_{1,1} = 90$.



Matrix Multiplication (Contd.)

Maximize Parallelism

One way is to compute each of the n^3 operations in a separate processor, requiring n^3 processors. Then, for each C_{ij} (a total of n^2 processors), we can get its sum by collecting the n products that correspond to it in $n - 1$ additional steps. This gives us a total of n arithmetic operations using n^3 processors. Can we do better?
Yes! We can use binary trees for our addition. With each step, we cut the number of additions by half, giving us $\log n$ steps using n^3 processors.

Example

We want to add 1, 10, 27, and 52 to get $C_{1,1} = 90$.

$$\begin{tikzcd}
1 \arrow[dr] & & 10 \arrow[dl] & & 27 \arrow[dr] & & 52 \arrow[dl] \\
& 11 \arrow[drr] & & & & 79 \arrow[dll] & \\
& & & 90 & & &
\end{tikzcd}$$

Matrix Multiplication (Contd.)

Key Observation

Our goal in parallel algorithms is to achieve a logarithmic, or polylogarithmic, time using polynomial processors. This gives us an exponential drop in the complexity. The other thing to consider is that the amount of work done by a parallel algorithm can be no smaller than the time complexity of the best sequential algorithm.

Brent's Principle

If the amount of work needed is $c_1 n^i$ and the parallel time is $c_2 \log^j n$, we only require $\frac{n^i}{\log n}$ processors.

Matrix Multiplication (Contd.)

Key Observation

Our goal in parallel algorithms is to achieve a logarithmic, or polylogarithmic, time using polynomial processors. This gives us an exponential drop in the complexity. The other thing to consider is that the amount of work done by a parallel algorithm can be no smaller than the time complexity of the best sequential algorithm.

Brent's Principle

If the amount of work needed is $c_1 n^i$ and the parallel time is $c_2 \log^j n$, we only require $\frac{n^i}{\log n}$ processors.

Matrix Multiplication (Contd.)

Key Observation

Our goal in parallel algorithms is to achieve a logarithmic, or polylogarithmic, time using polynomial processors. This gives us an exponential drop in the complexity. The other thing to consider is that the amount of work done by a parallel algorithm can be no smaller than the time complexity of the best sequential algorithm.

Brent's Principle

If the amount of work needed is $c_1 n^i$ and the parallel time is $c_2 \log^j n$, we only require $\frac{n^i}{\log n}$ processors.

Matrix Multiplication (Contd.)

Key Observation

Our goal in parallel algorithms is to achieve a logarithmic, or polylogarithmic, time using polynomial processors. This gives us an exponential drop in the complexity. The other thing to consider is that the amount of work done by a parallel algorithm can be no smaller than the time complexity of the best sequential algorithm.

Brent's Principle

If the amount of work needed is $c_1 n^i$ and the parallel time is $c_2 \log^j n$, we only require $\frac{n^i}{\log n}$ processors.

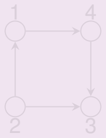
Outline

Reachability

Problem

Given a graph $G = (V, E)$ and two nodes $1, n \in V$, is there a path from 1 to n?

Example



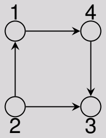
Example: Is node 3 reachable from node 1?

Reachability

Problem

Given a graph $G = (V, E)$ and two nodes $1, n \in V$, is there a path from 1 to n?

Example



Example: Is node 3 reachable from node 1?

REACHABILITY (Contd.)

Approach

We cannot parallelize the sequential algorithm because the number of parallel steps will be at least equal to the shortest path from the start node to the goal node, and this path can be as long as $n - 1$. In fact, we have to *forget* everything we know about sequential algorithms to make this work.

We can use Matrix multiplication for this. Let A be the adjacency matrix of the graph, where we added the self-loops: $A_{ii} = 1\ \forall i$. Suppose we compute the Boolean product of A with itself $A^2 = A \cdot A$, where $A^2_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}$. Note that $A^2_{ij} = 1$ if and only if there is a path of length 2 or less from node i to node j. If we apply this to $A^{\lceil \log n \rceil}$, we get $A^{2\lceil \log n \rceil}$, which is the adjacency matrix of the transitive closure of A, which is simply the answers of all possible REACHABILITY instances on the graph. This can be computed in $O(\log^2 n)$ parallel steps with $O(n^3 \log n)$ total work and, by Brent's principle, $O(\frac{n^3}{\log n})$ processors.

REACHABILITY (Contd.)

Approach

We cannot parallelize the sequential algorithm because the number of parallel steps will be at least equal to the shortest path from the start node to the goal node, and this path can be as long as $n - 1$. In fact, we have to *forget* everything we know about sequential algorithms to make this work.
We can use Matrix multiplication for this. Let A be the adjacency matrix of the graph, where we added the self-loops: $A_{ii} = 1 \; \forall i$. Suppose we compute the Boolean product of A with itself $A^2 = A \cdot A$, where $A^2_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}$. Note that $A^2_{ij} = 1$ if and only if there is a path of length 2 or less from node i to node j. If we apply this to $A^{\lceil \log n \rceil}$, we get $A^{2^{\lceil \log n \rceil}}$, which is the adjacency matrix of the transitive closure of A, which is simply the answers of all possible REACHABILITY instances on the graph. This can be computed in $O(\log^2 n)$ parallel steps with $O(n^3 \log n)$ total work and, by Brent's principle, $O(\frac{n^3}{\log n})$ processors.

REACHABILITY (Contd.)

Approach

We cannot parallelize the sequential algorithm because the number of parallel steps will be at least equal to the shortest path from the start node to the goal node, and this path can be as long as $n-1$. In fact, we have to *forget* everything we know about sequential algorithms to make this work.
We can use Matrix multiplication for this. Let A be the adjacency matrix of the graph, where we added the self-loops: $A_{ii} = 1\ \forall i$. Suppose we compute the Boolean product of A with itself $A^2 = A \cdot A$, where $A^2_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}$. Note that $A^2_{ij} = 1$ if and only if there is a path of length 2 or less from node i to node j. If we apply this to $A^{\lceil \log n \rceil}$, we get $A^{2^{\lceil \log n \rceil}}$, which is the adjacency matrix of the transitive closure of A, which is simply the answers of all possible REACHABILITY instances on the graph. This can be computed in $O(\log^2 n)$ parallel steps with $O(n^3 \log n)$ total work and, by Brent's principle, $O(\frac{n^3}{\log n})$ processors.

REACHABILITY (Contd.)

Approach

We cannot parallelize the sequential algorithm because the number of parallel steps will be at least equal to the shortest path from the start node to the goal node, and this path can be as long as $n - 1$. In fact, we have to *forget* everything we know about sequential algorithms to make this work.
We can use Matrix multiplication for this. Let A be the adjacency matrix of the graph, where we added the self-loops: $A_{ii} = 1\ \forall i$. Suppose we compute the Boolean product of A with itself $A^2 = A \cdot A$, where $A^2_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}$. Note that $A^2_{ij} = 1$ if and only if there is a path of length 2 or less from node i to node j. If we apply this to $A^{\lceil \log n \rceil}$, we get $A^{2 \lceil \log n \rceil}$, which is the adjacency matrix of the transitive closure of A, which is simply the answers of all possible REACHABILITY instances on the graph. This can be computed in $O(\log^2 n)$ parallel steps with $O(n^3 \log n)$ total work and, by Brent's principle, $O(\frac{n^3}{\log n})$ processors.

REACHABILITY (Contd.)

Approach

We cannot parallelize the sequential algorithm because the number of parallel steps will be at least equal to the shortest path from the start node to the goal node, and this path can be as long as $n - 1$. In fact, we have to *forget* everything we know about sequential algorithms to make this work.

We can use Matrix multiplication for this. Let A be the adjacency matrix of the graph, where we added the self-loops: $A_{ii} = 1 \ \forall i$. Suppose we compute the Boolean product of A with itself $A^2 = A \cdot A$, where $A^2_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}$. Note that $A^2_{ij} = 1$ if and only if there is a path of length 2 or less from node i to node j. If we apply this to $A^{\lceil \log n \rceil}$, we get $A^{2^{\lceil \log n \rceil}}$, which is the adjacency matrix of the transitive closure of A, which is simply the answers of all possible REACHABILITY instances on the graph. This can be computed in $O(\log^2 n)$ parallel steps with $O(n^3 \log n)$ total work and, by Brent's principle, $O(\frac{n^3}{\log n})$ processors.

REACHABILITY (Contd.)

Approach

We cannot parallelize the sequential algorithm because the number of parallel steps will be at least equal to the shortest path from the start node to the goal node, and this path can be as long as $n - 1$. In fact, we have to *forget* everything we know about sequential algorithms to make this work.
We can use Matrix multiplication for this. Let A be the adjacency matrix of the graph, where we added the self-loops: $A_{ii} = 1 \ \forall i$. Suppose we compute the Boolean product of A with itself $A^2 = A \cdot A$, where $A^2_{ij} = \displaystyle\bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}$. Note that $A^2_{ij} = 1$ if and only if there is a path of length 2 or less from node i to node j. If we apply this to $A^{\lceil \log n \rceil}$, we get $A^{2^{\lceil \log n \rceil}}$, which is the adjacency matrix of the transitive closure of A, which is simply the answers of all possible REACHABILITY instances on the graph. This can be computed in $O(\log^2 n)$ parallel steps with $O(n^3 \log n)$ total work and, by Brent's principle, $O(\frac{n^3}{\log n})$ processors.

REACHABILITY (Contd.)

Approach

We cannot parallelize the sequential algorithm because the number of parallel steps will be at least equal to the shortest path from the start node to the goal node, and this path can be as long as $n-1$. In fact, we have to *forget* everything we know about sequential algorithms to make this work.
We can use Matrix multiplication for this. Let A be the adjacency matrix of the graph, where we added the self-loops: $A_{ii} = 1\ \forall i$. Suppose we compute the Boolean product of A with itself $A^2 = A \cdot A$, where $A^2_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}$. Note that $A^2_{ij} = 1$ if and only if there is a path of length 2 or less from node i to node j. If we apply this to $A^{\lceil \log n \rceil}$, we get $A^{2^{\lceil \log n \rceil}}$, which is the adjacency matrix of the transitive closure of A, which is simply the answers of all possible REACHABILITY instances on the graph. This can be computed in $O(\log^2 n)$ parallel steps with $O(n^3 \log n)$ total work and, by Brent's principle, $O(\frac{n^3}{\log n})$ processors.

REACHABILITY (Contd.)

Approach

We cannot parallelize the sequential algorithm because the number of parallel steps will be at least equal to the shortest path from the start node to the goal node, and this path can be as long as $n-1$. In fact, we have to *forget* everything we know about sequential algorithms to make this work.
We can use Matrix multiplication for this. Let A be the adjacency matrix of the graph, where we added the self-loops: $A_{ii} = 1\ \forall i$. Suppose we compute the Boolean product of A with itself $A^2 = A \cdot A$, where $A^2_{ij} = \bigvee_{k=1}^{n} A_{ik} \wedge A_{kj}$. Note that $A^2_{ij} = 1$ if and only if there is a path of length 2 or less from node i to node j. If we apply this to $A^{\lceil \log n \rceil}$, we get $A^{2^{\lceil \log n \rceil}}$, which is the adjacency matrix of the transitive closure of A, which is simply the answers of all possible REACHABILITY instances on the graph. This can be computed in $O(\log^2 n)$ parallel steps with $O(n^3 \log n)$ total work and, by Brent's principle, $O(\frac{n^3}{\log n})$ processors.

Reachability (Contd.)

Example

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

After applying Matrix multiplication with the adjacency matrix, we can see that node 3 is reachable from node 1 with a path of length 2.

Reachability (Contd.)

Example

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

After applying Matrix multiplication with the adjacency matrix, we can see that node 3 is reachable from node 1 with a path of length 2.

Outline

Prefix Sums Problem

Problem

Given n integers $x_1, \ldots, x_n$, compute all of the sums of the form $\sum_{i=2}^{j} x_i$, where $j = 1, \ldots, n$.

Example

Suppose we are given the list $(1, 2, 3, 4, 5, 6, 7, 8)$. This is trivial to solve sequentially since we would start with $1 + 2$, then $1 + 2 + 3$, up until $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$ to get $(1, 3, 6, 10, 15, 21, 28, 36)$. However, this approach it too sequential to parallelize.

Prefix Sums Problem

Problem

Given n integers $x_1, ..., x_n$, compute all of the sums of the form $\sum_{i=2}^{j} x_i$, where $j = 1, ..., n$.

Example

Suppose we are given the list $(1, 2, 3, 4, 5, 6, 7, 8)$. This is trivial to solve sequentially since we would start with $1 + 2$, then $1 + 2 + 3$, up until $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$ to get $(1, 3, 6, 10, 15, 21, 28, 36)$. However, this approach it too sequential to parallelize.

Prefix Sums Problem

Problem

Given n integers $x_1, \ldots, x_n$, compute all of the sums of the form $\sum_{i=2}^{j} x_i$, where $j = 1, \ldots, n$.

Example

Suppose we are given the list $(1, 2, 3, 4, 5, 6, 7, 8)$. This is trivial to solve sequentially since we would start with $1 + 2$, then $1 + 2 + 3$, up until $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$ to get $(1, 3, 6, 10, 15, 21, 28, 36)$. However, this approach it too sequential to parallelize.

Prefix Sums Problem

Problem

Given n integers $x_1, \ldots, x_n$, compute all of the sums of the form $\sum_{i=2}^{j} x_i$, where $j = 1, \ldots, n$.

Example

Suppose we are given the list $(1, 2, 3, 4, 5, 6, 7, 8)$. This is trivial to solve sequentially since we would start with $1 + 2$, then $1 + 2 + 3$, up until $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8$ to get $(1, 3, 6, 10, 15, 21, 28, 36)$. However, this approach it too sequential to parallelize.

Prefix Sums Problem (Contd.)

Algorithm

We can use recursion for this algorithm. We assume that n is a power of 2; otherwise, we add harmless elements (such as 0). Our first parallel step would be to get the sums of $(x_1 + x_2), (x_3 + x_4), \ldots, (x_{n-1} + x_n)$. We then get the prefix sums of this sequence using recursion, which gives us the even numbered items in the list.
Using our example, we would have $(3, 7, 11, 15)$ after the first parallel step, and then use recursion to get $(3, 10, 21, 36)$.
We now use these numbers and our original list to get the rest of the values using one more parallel addition step. With our example, we get the values $(1, 3+3=6, 10+5=15, 21+7=28)$. The total number of parallel steps is $2\log n$, and the amount of work needed is $n + \frac{n}{2} + \frac{n}{4} + \ldots \leq 2n$, which, by Brent's principle, requires $\frac{n}{\log n}$ processors.

Prefix Sums Problem (Contd.)

Algorithm

We can use recursion for this algorithm. We assume that n is a power of 2; otherwise, we add harmless elements (such as 0). Our first parallel step would be to get the sums of $(x_1 + x_2), (x_3 + x_4), \ldots, (x_{n-1} + x_n)$. We then get the prefix sums of this sequence using recursion, which gives us the even numbered items in the list.
Using our example, we would have $(3, 7, 11, 15)$ after the first parallel step, and then use recursion to get $(3, 10, 21, 36)$.
We now use these numbers and our original list to get the rest of the values using one more parallel addition step. With our example, we get the values $(1, 3 + 3 = 6, 10 + 5 = 15, 21 + 7 = 28)$. The total number of parallel steps is $2 \log n$, and the amount of work needed is $n + \frac{n}{2} + \frac{n}{4} + \ldots \leq 2n$, which, by Brent's principle, requires $\frac{n}{\log n}$ processors.

Prefix Sums Problem (Contd.)

Algorithm

We can use recursion for this algorithm. We assume that n is a power of 2; otherwise, we add harmless elements (such as 0). Our first parallel step would be to get the sums of $(x_1 + x_2), (x_3 + x_4), \ldots, (x_{n-1} + x_n)$. We then get the prefix sums of this sequence using recursion, which gives us the even numbered items in the list.

Using our example, we would have $(3, 7, 11, 15)$ after the first parallel step, and then use recursion to get $(3, 10, 21, 36)$.

We now use these numbers and our original list to get the rest of the values using one more parallel addition step. With our example, we get the values $(1, 3+3=6, 10+5=15, 21+7=28)$. The total number of parallel steps is $2\log n$, and the amount of work needed is $n + \frac{n}{2} + \frac{n}{4} + \ldots \leq 2n$, which, by Brent's principle, requires $\frac{n}{\log n}$ processors.

Prefix Sums Problem (Contd.)

Algorithm

We can use recursion for this algorithm. We assume that n is a power of 2; otherwise, we add harmless elements (such as 0). Our first parallel step would be to get the sums of $(x_1 + x_2), (x_3 + x_4), \ldots, (x_{n-1} + x_n)$. We then get the prefix sums of this sequence using recursion, which gives us the even numbered items in the list.
Using our example, we would have $(3, 7, 11, 15)$ after the first parallel step, and then use recursion to get $(3, 10, 21, 36)$.
We now use these numbers and our original list to get the rest of the values using one more parallel addition step. With our example, we get the values $(1, 3 + 3 = 6, 10 + 5 = 15, 21 + 7 = 28)$. The total number of parallel steps is $2 \log n$, and the amount of work needed is $n + \frac{n}{2} + \frac{n}{4} + \ldots \leq 2n$, which, by Brent's principle, requires $\frac{n}{\log n}$ processors.

Prefix Sums Problem (Contd.)

Algorithm

We can use recursion for this algorithm. We assume that n is a power of 2; otherwise, we add harmless elements (such as 0). Our first parallel step would be to get the sums of $(x_1 + x_2), (x_3 + x_4), \ldots, (x_{n-1} + x_n)$. We then get the prefix sums of this sequence using recursion, which gives us the even numbered items in the list.
Using our example, we would have $(3, 7, 11, 15)$ after the first parallel step, and then use recursion to get $(3, 10, 21, 36)$.
We now use these numbers and our original list to get the rest of the values using one more parallel addition step. With our example, we get the values $(1, 3 + 3 = 6, 10 + 5 = 15, 21 + 7 = 28)$. The total number of parallel steps is $2 \log n$, and the amount of work needed is $n + \frac{n}{2} + \frac{n}{4} + \ldots \leq 2n$, which, by Brent's principle, requires $\frac{n}{\log n}$ processors.

Prefix Sums Problem (Contd.)

Algorithm

We can use recursion for this algorithm. We assume that n is a power of 2; otherwise, we add harmless elements (such as 0). Our first parallel step would be to get the sums of $(x_1 + x_2), (x_3 + x_4), \ldots, (x_{n-1} + x_n)$. We then get the prefix sums of this sequence using recursion, which gives us the even numbered items in the list.
Using our example, we would have $(3, 7, 11, 15)$ after the first parallel step, and then use recursion to get $(3, 10, 21, 36)$.
We now use these numbers and our original list to get the rest of the values using one more parallel addition step. With our example, we get the values $(1, 3 + 3 = 6, 10 + 5 = 15, 21 + 7 = 28)$. The total number of parallel steps is $2 \log n$, and the amount of work needed is $n + \frac{n}{2} + \frac{n}{4} + \ldots \leq 2n$, which, by Brent's principle, requires $\frac{n}{\log n}$ processors.

Prefix Sums Problem (Contd.)

Algorithm

We can use recursion for this algorithm. We assume that n is a power of 2; otherwise, we add harmless elements (such as 0). Our first parallel step would be to get the sums of $(x_1 + x_2), (x_3 + x_4), \ldots, (x_{n-1} + x_n)$. We then get the prefix sums of this sequence using recursion, which gives us the even numbered items in the list.
Using our example, we would have $(3, 7, 11, 15)$ after the first parallel step, and then use recursion to get $(3, 10, 21, 36)$.
We now use these numbers and our original list to get the rest of the values using one more parallel addition step. With our example, we get the values $(1, 3 + 3 = 6, 10 + 5 = 15, 21 + 7 = 28)$. The total number of parallel steps is $2 \log n$, and the amount of work needed is $n + \frac{n}{2} + \frac{n}{4} + \ldots \le 2n$, which, by Brent's principle, requires $\frac{n}{\log n}$ processors.

Prefix Sums Problem (Contd.)

Algorithm

We can use recursion for this algorithm. We assume that n is a power of 2; otherwise, we add harmless elements (such as 0). Our first parallel step would be to get the sums of $(x_1 + x_2), (x_3 + x_4), ..., (x_{n-1} + x_n)$. We then get the prefix sums of this sequence using recursion, which gives us the even numbered items in the list.
Using our example, we would have $(3, 7, 11, 15)$ after the first parallel step, and then use recursion to get $(3, 10, 21, 36)$.
We now use these numbers and our original list to get the rest of the values using one more parallel addition step. With our example, we get the values $(1, 3 + 3 = 6, 10 + 5 = 15, 21 + 7 = 28)$. The total number of parallel steps is $2 \log n$, and the amount of work needed is $n + \frac{n}{2} + \frac{n}{4} + ... \leq 2n$, which, by Brent's principle, requires $\frac{n}{\log n}$ processors.

Outline

Determinants

Problem

Given a matrix A, find its determinant.

Matrix Inversion

We can merge this problem with matrix inversion and then solve both. Suppose we are given a matrix A, and let $A[i]$ be the matrix omitting the first $n - i$ rows in columns (i.e. $A[i]$ is the i x i lower right-hand corner of A). Consider the inverse $A[i]^{-1}$ and the first element $(A[i]^{-1})_{11}$. According to Cramer's rule (which can be used to solve a system of two equations with two variables), we get $(A[i]^{-1})_{11} = \frac{\det A[i-1]}{\det A[i]}$, which holds for $i = n, n-1, ..., 2$. Back-solving these equations, and since $A[n] = A$, we get

$$\det A = (\prod_{i=1}^{n} (A[i]^{-1})_{11})^{-1}.$$

Determinants

Problem

Given a matrix A, find its determinant.

Matrix Inversion

We can merge this problem with matrix inversion and then solve both. Suppose we are given a matrix A, and let $A[i]$ be the matrix omitting the first $n - i$ rows in columns (i.e. $A[i]$ is the i x i lower right-hand corner of A). Consider the inverse $A[i]^{-1}$ and the first element $(A[i]^{-1})_{11}$. According to Cramer's rule (which can be used to solve a system of two equations with two variables), we get $(A[i]^{-1})_{11} = \frac{\det A[i-1]}{\det A[i]}$, which holds for $i = n, n-1, ..., 2$. Back-solving these equations, and since $A[n] = A$, we get

$$\det A = (\prod_{i=1}^{n}(A[i]^{-1})_{11})^{-1}.$$

Determinants

Problem

Given a matrix A, find its determinant.

Matrix Inversion

We can merge this problem with matrix inversion and then solve both. Suppose we are given a matrix A, and let $A[i]$ be the matrix omitting the first $n - i$ rows in columns (i.e. $A[i]$ is the i x i lower right-hand corner of A). Consider the inverse $A[i]^{-1}$ and the first element $(A[i]^{-1})_{11}$. According to Cramer's rule (which can be used to solve a system of two equations with two variables), we get $(A[i]^{-1})_{11} = \frac{\det A[i-1]}{\det A[i]}$, which holds for $i = n, n-1, ..., 2$. Back-solving these equations, and since $A[n] = A$, we get

$\det A = (\prod_{i=1}^{n} (A[i]^{-1})_{11})^{-1}$.

Determinants

Problem

Given a matrix A, find its determinant.

Matrix Inversion

We can merge this problem with matrix inversion and then solve both. Suppose we are given a matrix A, and let $A[i]$ be the matrix omitting the first $n - i$ rows in columns (i.e. $A[i]$ is the i x i lower right-hand corner of A). Consider the inverse $A[i]^{-1}$ and the first element $(A[i]^{-1})_{11}$. According to Cramer's rule (which can be used to solve a system of two equations with two variables), we get $(A[i]^{-1})_{11} = \frac{\det A[i-1]}{\det A[i]}$, which holds for $i = n, n-1, ..., 2$. Back-solving these equations, and since $A[n] = A$, we get

$$\det A = (\prod_{i=1}^{n} (A[i]^{-1})_{11})^{-1}.$$

Determinants

Problem

Given a matrix A, find its determinant.

Matrix Inversion

We can merge this problem with matrix inversion and then solve both. Suppose we are given a matrix A, and let $A[i]$ be the matrix omitting the first $n - i$ rows in columns (i.e. $A[i]$ is the i x i lower right-hand corner of A). Consider the inverse $A[i]^{-1}$ and the first element $(A[i]^{-1})_{11}$. According to Cramer's rule (which can be used to solve a system of two equations with two variables), we get $(A[i]^{-1})_{11} = \frac{\det A[i-1]}{\det A[i]}$, which holds for $i = n, n-1, ..., 2$. Back-solving these equations, and since $A[n] = A$, we get

$$\det A = (\prod_{i=1}^{n}(A[i]^{-1})_{11})^{-1}.$$

Determinants

Problem

Given a matrix A, find its determinant.

Matrix Inversion

We can merge this problem with matrix inversion and then solve both. Suppose we are given a matrix A, and let $A[i]$ be the matrix omitting the first $n - i$ rows in columns (i.e. $A[i]$ is the i x i lower right-hand corner of A). Consider the inverse $A[i]^{-1}$ and the first element $(A[i]^{-1})_{11}$. According to Cramer's rule (which can be used to solve a system of two equations with two variables), we get $(A[i]^{-1})_{11} = \frac{\det A[i-1]}{\det A[i]}$, which holds for $i = n, n-1, ..., 2$. Back-solving these equations, and since $A[n] = A$, we get

$$\det A = (\prod_{i=1}^{n} (A[i]^{-1})_{11})^{-1}.$$

Determinants (Contd.)

Approach

We will compute the determinants by first computing the inverses of many matrices, all in parallel, then multiplying the upper-left entries, and finally inverting the result.
However, we need to use a symbolic matrix for this to work; specifically, the matrix $I - xA$. This is because we have a similar situation for 1 x 1 matrices, where we get the formal power series $(1 - xA)^{-1} = \sum_{i=0}^{\infty}(xA)^i$. In order to get $(I - xA[i])^{-1}$, we only have to compute and add in the parallel power of $xA[i]$ using prefix sums.
What about the infinite summation in the previous formula? Since we only need to compute the determinant of $I - xA$, which is a polynomial in x of degree n, we can stop using the summation at the nth addend. Therefore, we compute in parallel all $(I - xA)[i]^{-1}$s, each by computing by parallel prefix all matrices of the form $(xA)^i$ mod x^{n+1}, and then adding them together.

Determinants (Contd.)

Approach

We will compute the determinants by first computing the inverses of many matrices, all in parallel, then multiplying the upper-left entries, and finally inverting the result. However, we need to use a symbolic matrix for this to work; specifically, the matrix $I - xA$. This is because we have a similar situation for 1 x 1 matrices, where we get the formal power series $(1 - xA)^{-1} = \sum_{i=0}^{\infty}(xA)^i$. In order to get $(I - xA[i])^{-1}$, we only have to compute and add in the parallel power of $xA[i]$ using prefix sums.
What about the infinite summation in the previous formula? Since we only need to compute the determinant of $I - xA$, which is a polynomial in x of degree n, we can stop using the summation at the nth addend. Therefore, we compute in parallel all $(I - xA)[i]^{-1}$s, each by computing by parallel prefix all matrices of the form $(xA)^i \bmod x^{n+1}$, and then adding them together.

Determinants (Contd.)

Approach

We will compute the determinants by first computing the inverses of many matrices, all in parallel, then multiplying the upper-left entries, and finally inverting the result. However, we need to use a symbolic matrix for this to work; specifically, the matrix $I - xA$. This is because we have a similar situation for 1 x 1 matrices, where we get the formal power series $(1 - xA)^{-1} = \sum_{i=0}^{\infty}(xA)^i$. In order to get $(I - xA[i])^{-1}$, we only have to compute and add in the parallel power of $xA[i]$ using prefix sums.
What about the infinite summation in the previous formula? Since we only need to compute the determinant of $I - xA$, which is a polynomial in x of degree n, we can stop using the summation at the nth addend. Therefore, we compute in parallel all $(I - xA)[i]^{-1}$s, each by computing by parallel prefix all matrices of the form $(xA)^i$ mod x^{n+1}, and then adding them together.

Determinants (Contd.)

Approach

We will compute the determinants by first computing the inverses of many matrices, all in parallel, then multiplying the upper-left entries, and finally inverting the result. However, we need to use a symbolic matrix for this to work; specifically, the matrix $I - xA$. This is because we have a similar situation for 1 x 1 matrices, where we get the formal power series $(1 - xA)^{-1} = \sum_{i=0}^{\infty} (xA)^i$. In order to get $(I - xA[i])^{-1}$, we only have to compute and add in the parallel power of $xA[i]$ using prefix sums.
What about the infinite summation in the previous formula? Since we only need to compute the determinant of $I - xA$, which is a polynomial in x of degree n, we can stop using the summation at the nth addend. Therefore, we compute in parallel all $(I - xA)[i]^{-1}$s, each by computing by parallel prefix all matrices of the form $(xA)^i$ mod x^{n+1}, and then adding them together.

Determinants (Contd.)

Approach

We will compute the determinants by first computing the inverses of many matrices, all in parallel, then multiplying the upper-left entries, and finally inverting the result. However, we need to use a symbolic matrix for this to work; specifically, the matrix $I - xA$. This is because we have a similar situation for 1 x 1 matrices, where we get the formal power series $(1 - xA)^{-1} = \sum_{i=0}^{\infty} (xA)^i$. In order to get $(I - xA[i])^{-1}$, we only have to compute and add in the parallel power of $xA[i]$ using prefix sums.
What about the infinite summation in the previous formula? Since we only need to compute the determinant of $I - xA$, which is a polynomial in x of degree n, we can stop using the summation at the nth addend. Therefore, we compute in parallel all $(I - xA)[i]^{-1}$s, each by computing by parallel prefix all matrices of the form $(xA)^i \bmod x^{n+1}$, and then adding them together.

Determinants (Contd.)

Approach

We will compute the determinants by first computing the inverses of many matrices, all in parallel, then multiplying the upper-left entries, and finally inverting the result. However, we need to use a symbolic matrix for this to work; specifically, the matrix $I - xA$. This is because we have a similar situation for 1 x 1 matrices, where we get the formal power series $(1 - xA)^{-1} = \sum_{i=0}^{\infty}(xA)^i$. In order to get $(I - xA[i])^{-1}$, we only have to compute and add in the parallel power of $xA[i]$ using prefix sums.
What about the infinite summation in the previous formula? Since we only need to compute the determinant of $I - xA$, which is a polynomial in x of degree n, we can stop using the summation at the nth addend. Therefore, we compute in parallel all $(I - xA)[i]^{-1}$s, each by computing by parallel prefix all matrices of the form $(xA)^i$ mod x^{n+1}, and then adding them together.

Determinants (Contd.)

Approach

We will compute the determinants by first computing the inverses of many matrices, all in parallel, then multiplying the upper-left entries, and finally inverting the result. However, we need to use a symbolic matrix for this to work; specifically, the matrix $I - xA$. This is because we have a similar situation for 1 x 1 matrices, where we get the formal power series $(1 - xA)^{-1} = \sum_{i=0}^{\infty}(xA)^i$. In order to get $(I - xA[i])^{-1}$, we only have to compute and add in the parallel power of $xA[i]$ using prefix sums.
What about the infinite summation in the previous formula? Since we only need to compute the determinant of $I - xA$, which is a polynomial in x of degree n, we can stop using the summation at the nth addend. Therefore, we compute in parallel all $(I - xA)[i]^{-1}$s, each by computing by parallel prefix all matrices of the form $(xA)^i \bmod x^{n+1}$, and then adding them together.

Determinants (Contd.)

Approach (Contd.)

Once we have all $(I - xA)[i]^{-1}$s, we obtain the upper-left elements and multiply them together modulo x^{n+1} to obtain a polynomial of degree n in x, called $c_0(1 + xp(x))$ where $c_0 \neq 0$. This polynomial is the inverse of $\det(I - xA)$. Therefore, we can get the inverse of this by using the power series for inversion and truncate after the x^n term to get $(c_0(1 + xp(x)))^{-1} = \frac{1}{c_0} \sum_{i=1}^{\infty} (-xp(x))^i \bmod x^{n+1}$.

To get $\det A$, we simply get the coefficient of x^n in $\det(I - xA)$ if n is even. If n is odd, we multiply by -1.

Determinants (Contd.)

Approach (Contd.)

Once we have all $(I - xA)[i]^{-1}$s, we obtain the upper-left elements and multiply them together modulo x^{n+1} to obtain a polynomial of degree n in x, called $c_0(1 + xp(x))$ where $c_0 \neq 0$. This polynomial is the inverse of $\det(I - xA)$. Therefore, we can get the inverse of this by using the power series for inversion and truncate after the x^n term to get $(c_0(1 + xp(x)))^{-1} = \frac{1}{c_0} \sum_{i=1}^{\infty} (-xp(x))^i \bmod x^{n+1}$.

To get $\det A$, we simply get the coefficient of x^n in $\det(I - xA)$ if n is even. If n is odd, we multiply by -1.

Determinants (Contd.)

Approach (Contd.)

Once we have all $(I - xA)[i]^{-1}$s, we obtain the upper-left elements and multiply them together modulo x^{n+1} to obtain a polynomial of degree n in x, called $c_0(1 + xp(x))$ where $c_0 \neq 0$. This polynomial is the inverse of $\det(I - xA)$. Therefore, we can get the inverse of this by using the power series for inversion and truncate after the x^n term to get $(c_0(1 + xp(x)))^{-1} = \frac{1}{c_0} \sum_{i=1}^{\infty} (-xp(x))^i \bmod x^{n+1}$.

To get $\det A$, we simply get the coefficient of x^n in $\det(I - xA)$ if n is even. If n is odd, we multiply by -1.

Determinants (Contd.)

Approach (Contd.)

Once we have all $(I - xA)[i]^{-1}$s, we obtain the upper-left elements and multiply them together modulo x^{n+1} to obtain a polynomial of degree n in x, called $c_0(1 + xp(x))$ where $c_0 \neq 0$. This polynomial is the inverse of $\det(I - xA)$. Therefore, we can get the inverse of this by using the power series for inversion and truncate after the x^n term to get $(c_0(1 + xp(x)))^{-1} = \frac{1}{c_0} \sum_{i=1}^{\infty} (-xp(x))^i \bmod x^{n+1}$.

To get $\det A$, we simply get the coefficient of x^n in $\det(I - xA)$ if n is even. If n is odd, we multiply by -1.

Determinants (Contd.)

Approach (Contd.)

Once we have all $(I - xA)[i]^{-1}$s, we obtain the upper-left elements and multiply them together modulo x^{n+1} to obtain a polynomial of degree n in x, called $c_0(1 + xp(x))$ where $c_0 \neq 0$. This polynomial is the inverse of $\det(I - xA)$. Therefore, we can get the inverse of this by using the power series for inversion and truncate after the x^n term to get $(c_0(1 + xp(x)))^{-1} = \frac{1}{c_0} \sum_{i=1}^{\infty} (-xp(x))^i \bmod x^{n+1}$.

To get $\det A$, we simply get the coefficient of x^n in $\det(I - xA)$ if n is even. If n is odd, we multiply by -1.

Determinants (Contd.)

Example

Suppose we want to get the determinant of

$$A = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix}.$$

Using this method, we start with

$$I - xA = \begin{pmatrix} 1-x & -2x \\ x & 1-3x \end{pmatrix},$$

and we must compute the $(I - xA)[i]_{11}^{-1}$s for $i = 1, 2$.

Matrix $xA[1]$ is just $(3x)$, which means $\sum_{i=0}^{\infty}(xA[1])^i \bmod x^3 = (1 + 3x + 9x^2)$. This means the upper-left elements of this matrix is $(1 + 3x + 9x^2)$.

Determinants (Contd.)

Example

Suppose we want to get the determinant of

$$A = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix}.$$

Using this method, we start with

$$I - xA = \begin{pmatrix} 1-x & -2x \\ x & 1-3x \end{pmatrix},$$

and we must compute the $(I - xA)[i]_{11}^{-1}$s for $i = 1, 2$.

Matrix $xA[1]$ is just $(3x)$, which means $\sum_{i=0}^{\infty} (xA[1])^i \bmod x^3 = (1 + 3x + 9x^2)$. This means the upper-left elements of this matrix is $(1 + 3x + 9x^2)$.

Determinants (Contd.)

Example

Suppose we want to get the determinant of

$$A = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix}.$$

Using this method, we start with

$$I - xA = \begin{pmatrix} 1 - x & -2x \\ x & 1 - 3x \end{pmatrix},$$

and we must compute the $(I - xA)[i]_{11}^{-1}$s for $i = 1, 2$.

Matrix $xA[1]$ is just $(3x)$, which means $\sum_{i=0}^{\infty} (xA[1])^i \bmod x^3 = (1 + 3x + 9x^2)$. This means the upper-left elements of this matrix is $(1 + 3x + 9x^2)$.

Determinants (Contd.)

Example

Suppose we want to get the determinant of

$$A = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix}.$$

Using this method, we start with

$$I - xA = \begin{pmatrix} 1-x & -2x \\ x & 1-3x \end{pmatrix},$$

and we must compute the $(I - xA)[i]_{11}^{-1}$s for $i = 1, 2$.

Matrix $xA[1]$ is just $(3x)$, which means $\sum_{i=0}^{\infty}(xA[1])^i \bmod x^3 = (1 + 3x + 9x^2)$. This means the upper-left elements of this matrix is $(1 + 3x + 9x^2)$.

Determinants (Contd.)

Example

For $(I - xA)[2]^{-1}$, we need the powers

$$(xA[2])^0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, (xA[2])^1 = \begin{pmatrix} x & 2x \\ -x & 3x \end{pmatrix}, (xA[2])^2 = \begin{pmatrix} -x^2 & 8x^2 \\ -4x^2 & 7x^2 \end{pmatrix};$$

all higher powers will be ignored since we are working modulo x^3. Adding those together we get that

$(I - xA)[2]^{-1} = \begin{pmatrix} 1+x-x^2 & 2x+8x^2 \\ -x-4x^2 & 1+3x+7x^2 \end{pmatrix}$ mod x^3, and thus $((I - xA)[2]^{-1})_{11} = 1 + x - x^2$. Multiplying $(I - xA[1]^{-1})_{11}$ times $((I - xA)[2]^{-1})_{11}$ gives us $(1 + 3x + 9x^2)(1 + x - x^2) = 1 + 4x + 11x^2 = 1 + x(4 + 11x)$ mod x^3. We now invert this polynomial modulo x^3, which gives us $1 - (4x + 11x^2) + (4x + 11x^2)^2 = 1 - 4x + 5x^2$ mod x^3. Since the determinant of A is the coefficient of x^2, we see that $\det A = 5$.

Determinants (Contd.)

Example

For $(I - xA)[2]^{-1}$, we need the powers

$$(xA[2])^0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, (xA[2])^1 = \begin{pmatrix} x & 2x \\ -x & 3x \end{pmatrix}, (xA[2])^2 = \begin{pmatrix} -x^2 & 8x^2 \\ -4x^2 & 7x^2 \end{pmatrix};$$

all higher powers will be ignored since we are working modulo x^3. Adding those together we get that
$(I - xA)[2]^{-1} = \begin{pmatrix} 1 + x - x^2 & 2x + 8x^2 \\ -x - 4x^2 & 1 + 3x + 7x^2 \end{pmatrix}$ mod x^3, and thus
$((I - xA)[2]^{-1})_{11} = 1 + x - x^2$. Multiplying $(I - xA[1]^{-1})_{11}$ times $((I - xA)[2]^{-1})_{11}$ gives us $(1 + 3x + 9x^2)(1 + x - x^2) = 1 + 4x + 11x^2 = 1 + x(4 + 11x)$ mod x^3.
We now invert this polynomial modulo x^3, which gives us
$1 - (4x + 11x^2) + (4x + 11x^2)^2 = 1 - 4x + 5x^2$ mod x^3. Since the determinant of A is the coefficient of x^2, we see that $\det A = 5$.

Determinants (Contd.)

Example

For $(I - xA)[2]^{-1}$, we need the powers

$$(xA[2])^0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, (xA[2])^1 = \begin{pmatrix} x & 2x \\ -x & 3x \end{pmatrix}, (xA[2])^2 = \begin{pmatrix} -x^2 & 8x^2 \\ -4x^2 & 7x^2 \end{pmatrix};$$

all higher powers will be ignored since we are working modulo x^3. Adding those together we get that
$(I - xA)[2]^{-1} = \begin{pmatrix} 1 + x - x^2 & 2x + 8x^2 \\ -x - 4x^2 & 1 + 3x + 7x^2 \end{pmatrix}$ mod x^3, and thus
$((I - xA)[2]^{-1})_{11} = 1 + x - x^2$. Multiplying $(I - xA[1]^{-1})_{11}$ times $((I - xA)[2]^{-1})_{11}$ gives us $(1 + 3x + 9x^2)(1 + x - x^2) = 1 + 4x + 11x^2 = 1 + x(4 + 11x)$ mod x^3.
We now invert this polynomial modulo x^3, which gives us
$1 - (4x + 11x^2) + (4x + 11x^2)^2 = 1 - 4x + 5x^2$ mod x^3. Since the determinant of A is the coefficient of x^2, we see that $\det A = 5$.

Determinants (Contd.)

Example

For $(I - xA)[2]^{-1}$, we need the powers

$$(xA[2])^0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, (xA[2])^1 = \begin{pmatrix} x & 2x \\ -x & 3x \end{pmatrix}, (xA[2])^2 = \begin{pmatrix} -x^2 & 8x^2 \\ -4x^2 & 7x^2 \end{pmatrix};$$

all higher powers will be ignored since we are working modulo x^3. Adding those together we get that
$(I - xA)[2]^{-1} = \begin{pmatrix} 1 + x - x^2 & 2x + 8x^2 \\ -x - 4x^2 & 1 + 3x + 7x^2 \end{pmatrix}$ mod x^3, and thus
$((I - xA)[2]^{-1})_{11} = 1 + x - x^2$. Multiplying $(I - xA[1]^{-1})_{11}$ times $((I - xA)[2]^{-1})_{11}$ gives us $(1 + 3x + 9x^2)(1 + x - x^2) = 1 + 4x + 11x^2 = 1 + x(4 + 11x)$ mod x^3.
We now invert this polynomial modulo x^3, which gives us
$1 - (4x + 11x^2) + (4x + 11x^2)^2 = 1 - 4x + 5x^2$ mod x^3. Since the determinant of A is the coefficient of x^2, we see that $\det A = 5$.

Determinants (Contd.)

Example

For $(I - xA)[2]^{-1}$, we need the powers

$$(xA[2])^0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, (xA[2])^1 = \begin{pmatrix} x & 2x \\ -x & 3x \end{pmatrix}, (xA[2])^2 = \begin{pmatrix} -x^2 & 8x^2 \\ -4x^2 & 7x^2 \end{pmatrix};$$

all higher powers will be ignored since we are working modulo x^3. Adding those together we get that
$(I - xA)[2]^{-1} = \begin{pmatrix} 1 + x - x^2 & 2x + 8x^2 \\ -x - 4x^2 & 1 + 3x + 7x^2 \end{pmatrix}$ mod x^3, and thus
$((I - xA)[2]^{-1})_{11} = 1 + x - x^2$. Multiplying $(I - xA[1]^{-1})_{11}$ times $((I - xA)[2]^{-1})_{11}$ gives us $(1 + 3x + 9x^2)(1 + x - x^2) = 1 + 4x + 11x^2 = 1 + x(4 + 11x)$ mod x^3.
We now invert this polynomial modulo x^3, which gives us
$1 - (4x + 11x^2) + (4x + 11x^2)^2 = 1 - 4x + 5x^2$ mod x^3. Since the determinant of A is the coefficient of x^2, we see that $\det A = 5$.

Determinants (Contd.)

Note

Each of the three stages (computing inverses, multiplying corner elements, inverting result) can be done in $O(\log^2 n)$ parallel steps. The first stage needs n parallel matrix multiplications, or $O(n^4)$ total work. However, matrix elements are nth degree polynomials, not bits; meaning each operation on the polynomials can be done in $O(\log n)$ parallel arithmetic steps for $O(n^2)$ total work.

Now, if the elements of the matrix are b-bit integers, then the coefficients of the polynomials have $O(nb)$ bits, and each arithmetic operation takes $O(\log n + \log b)$ bit operations and $O(n^2b^2)$ total work. Therefore, we can compute the determinant of an n x n matrix with b-bit integer entries in parallel time $O(\log^3 n(\log n + \log b))$, and $O(n^8b^2)$ total work. This is still logarithmic time and polynomial work.

Determinants (Contd.)

Note

Each of the three stages (computing inverses, multiplying corner elements, inverting result) can be done in $O(\log^2 n)$ parallel steps. The first stage needs n parallel matrix multiplications, or $O(n^4)$ total work. However, matrix elements are nth degree polynomials, not bits; meaning each operation on the polynomials can be done in $O(\log n)$ parallel arithmetic steps for $O(n^2)$ total work.
Now, if the elements of the matrix are b-bit integers, then the coefficients of the polynomials have $O(nb)$ bits, and each arithmetic operation takes $O(\log n + \log b)$ bit operations and $O(n^2 b^2)$ total work. Therefore, we can compute the determinant of an n x n matrix with b-bit integer entries in parallel time $O(\log^3 n(\log n + \log b))$, and $O(n^8 b^2)$ total work. This is still logarithmic time and polynomial work.

Determinants (Contd.)

Note

Each of the three stages (computing inverses, multiplying corner elements, inverting result) can be done in $O(\log^2 n)$ *parallel steps. The first stage needs n parallel matrix multiplications, or* $O(n^4)$ *total work. However, matrix elements are nth degree polynomials, not bits; meaning each operation on the polynomials can be done in* $O(\log n)$ *parallel arithmetic steps for* $O(n^2)$ *total work.*

Now, if the elements of the matrix are b-bit integers, then the coefficients of the polynomials have $O(nb)$ *bits, and each arithmetic operation takes* $O(\log n + \log b)$ *bit operations and* $O(n^2 b^2)$ *total work. Therefore, we can compute the determinant of an n x n matrix with b-bit integer entries in parallel time* $O(\log^3 n(\log n + \log b))$*, and* $O(n^8 b^2)$ *total work. This is still logarithmic time and polynomial work.*

Determinants (Contd.)

Note

Each of the three stages (computing inverses, multiplying corner elements, inverting result) can be done in $O(\log^2 n)$ *parallel steps. The first stage needs n parallel matrix multiplications, or* $O(n^4)$ *total work. However, matrix elements are nth degree polynomials, not bits; meaning each operation on the polynomials can be done in* $O(\log n)$ *parallel arithmetic steps for* $O(n^2)$ *total work.*
Now, if the elements of the matrix are b-bit integers, then the coefficients of the polynomials have $O(nb)$ *bits, and each arithmetic operation takes* $O(\log n + \log b)$ *bit operations and* $O(n^2 b^2)$ *total work. Therefore, we can compute the determinant of an n x n matrix with b-bit integer entries in parallel time* $O(\log^3 n(\log n + \log b))$*, and* $O(n^8 b^2)$ *total work. This is still logarithmic time and polynomial work.*

Determinants (Contd.)

Note

Each of the three stages (computing inverses, multiplying corner elements, inverting result) can be done in $O(\log^2 n)$ *parallel steps. The first stage needs n parallel matrix multiplications, or* $O(n^4)$ *total work. However, matrix elements are nth degree polynomials, not bits; meaning each operation on the polynomials can be done in* $O(\log n)$ *parallel arithmetic steps for* $O(n^2)$ *total work.*
Now, if the elements of the matrix are b-bit integers, then the coefficients of the polynomials have $O(nb)$ *bits, and each arithmetic operation takes* $O(\log n + \log b)$ *bit operations and* $O(n^2 b^2)$ *total work. Therefore, we can compute the determinant of an n x n matrix with b-bit integer entries in parallel time* $O(\log^3 n(\log n + \log b))$*, and* $O(n^8 b^2)$ *total work. This is still logarithmic time and polynomial work.*

Determinants (Contd.)

Note

Each of the three stages (computing inverses, multiplying corner elements, inverting result) can be done in $O(\log^2 n)$ *parallel steps. The first stage needs n parallel matrix multiplications, or* $O(n^4)$ *total work. However, matrix elements are nth degree polynomials, not bits; meaning each operation on the polynomials can be done in* $O(\log n)$ *parallel arithmetic steps for* $O(n^2)$ *total work.*
Now, if the elements of the matrix are b-bit integers, then the coefficients of the polynomials have $O(nb)$ *bits, and each arithmetic operation takes* $O(\log n + \log b)$ *bit operations and* $O(n^2 b^2)$ *total work. Therefore, we can compute the determinant of an n x n matrix with b-bit integer entries in parallel time* $O(\log^3 n(\log n + \log b))$*, and* $O(n^8 b^2)$ *total work. This is still logarithmic time and polynomial work.*

Models

Note

We have already seen different models of computation such as the Turing machine, the multistring variant, the RAM, and the nondeterministic Turing machine. For parallel computations, we will be using the Boolean circuit. This is because it has no "program counter," so its computational activity may take place at many gates concurrently.

Definition

Let C be a Boolean circuit where the size of C is the total number of gates in it, and the depth of C is the number of nodes in the longest path in C. Let $C = (C_0, C_1, \ldots)$ be a uniform family of circuits, and let $f(n)$ and $g(n)$ be functions from the integers to the integers. We say that the parallel time of C is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. Furthermore, we say that the total work of C is at most $g(n)$ if for all $n \geq 0$ the size of C_n is at most $g(n)$. Finally, we let **PT/WK**$(f(n), g(n))$ be the class of all languages $L \subseteq \{0,1\}^*$ such that there is a uniform family of circuits C deciding L with $O(f(n))$ parallel time and $O(g(n))$ work.

Models

Note

We have already seen different models of computation such as the Turing machine, the multistring variant, the RAM, and the nondeterministic Turing machine. For parallel computations, we will be using the Boolean circuit. This is because it has no "program counter," so its computational activity may take place at many gates concurrently.

Definition

Let C be a Boolean circuit where the size of C is the total number of gates in it, and the depth of C is the number of nodes in the longest path in C. Let $C = (C_0, C_1, \ldots)$ be a uniform family of circuits, and let $f(n)$ and $g(n)$ be functions from the integers to the integers. We say that the parallel time of C is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. Furthermore, we say that the total work of C is at most $g(n)$ if for all $n \geq 0$ the size of C_n is at most $g(n)$. Finally, we let **PT/WK**$(f(n), g(n))$ be the class of all languages $L \subseteq \{0,1\}^*$ such that there is a uniform family of circuits C deciding L with $O(f(n))$ parallel time and $O(g(n))$ work.

Models

Note

We have already seen different models of computation such as the Turing machine, the multistring variant, the RAM, and the nondeterministic Turing machine. For parallel computations, we will be using the Boolean circuit. This is because it has no "program counter," so its computational activity may take place at many gates concurrently.

Definition

Let C be a Boolean circuit where the size of C is the total number of gates in it, and the depth of C is the number of nodes in the longest path in C. Let $C = (C_0, C_1, \ldots)$ be a uniform family of circuits, and let $f(n)$ and $g(n)$ be functions from the integers to the integers. We say that the parallel time of C is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. Furthermore, we say that the total work of C is at most $g(n)$ if for all $n \geq 0$ the size of C_n is at most $g(n)$. Finally, we let **PT/WK**$(f(n), g(n))$ be the class of all languages $L \subseteq \{0,1\}^*$ such that there is a uniform family of circuits C deciding L with $O(f(n))$ parallel time and $O(g(n))$ work.

Models

Note

We have already seen different models of computation such as the Turing machine, the multistring variant, the RAM, and the nondeterministic Turing machine. For parallel computations, we will be using the Boolean circuit. This is because it has no "program counter," so its computational activity may take place at many gates concurrently.

Definition

Let C be a Boolean circuit where the size of C is the total number of gates in it, and the depth of C is the number of nodes in the longest path in C. Let $C = (C_0, C_1, \ldots)$ be a uniform family of circuits, and let $f(n)$ and $g(n)$ be functions from the integers to the integers. We say that the parallel time of C is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. Furthermore, we say that the total work of C is at most $g(n)$ if for all $n \geq 0$ the size of C_n is at most $g(n)$. Finally, we let **PT/WK**$(f(n), g(n))$ be the class of all languages $L \subseteq \{0, 1\}^*$ such that there is a uniform family of circuits C deciding L with $O(f(n))$ parallel time and $O(g(n))$ work.

Models

Note

We have already seen different models of computation such as the Turing machine, the multistring variant, the RAM, and the nondeterministic Turing machine. For parallel computations, we will be using the Boolean circuit. This is because it has no "program counter," so its computational activity may take place at many gates concurrently.

Definition

Let C be a Boolean circuit where the size of C is the total number of gates in it, and the depth of C is the number of nodes in the longest path in C. Let $C = (C_0, C_1, \ldots)$ be a uniform family of circuits, and let $f(n)$ and $g(n)$ be functions from the integers to the integers. We say that the parallel time of C is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. Furthermore, we say that the total work of C is at most $g(n)$ if for all $n \geq 0$ the size of C_n is at most $g(n)$. Finally, we let **PT/WK**$(f(n), g(n))$ be the class of all languages $L \subseteq \{0, 1\}^*$ such that there is a uniform family of circuits C deciding L with $O(f(n))$ parallel time and $O(g(n))$ work.

Models

Note

We have already seen different models of computation such as the Turing machine, the multistring variant, the RAM, and the nondeterministic Turing machine. For parallel computations, we will be using the Boolean circuit. This is because it has no "program counter," so its computational activity may take place at many gates concurrently.

Definition

Let C be a Boolean circuit where the size of C is the total number of gates in it, and the depth of C is the number of nodes in the longest path in C. Let $C = (C_0, C_1, ...)$ be a uniform family of circuits, and let $f(n)$ and $g(n)$ be functions from the integers to the integers. We say that the parallel time of C is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. Furthermore, we say that the total work of C is at most $g(n)$ if for all $n \geq 0$ the size of C_n is at most $g(n)$. Finally, we let **PT/WK**$(f(n), g(n))$ be the class of all languages $L \subseteq \{0, 1\}^*$ such that there is a uniform family of circuits C deciding L with $O(f(n))$ parallel time and $O(g(n))$ work.

Models

Note

We have already seen different models of computation such as the Turing machine, the multistring variant, the RAM, and the nondeterministic Turing machine. For parallel computations, we will be using the Boolean circuit. This is because it has no "program counter," so its computational activity may take place at many gates concurrently.

Definition

Let C be a Boolean circuit where the size of C is the total number of gates in it, and the depth of C is the number of nodes in the longest path in C. Let $C = (C_0, C_1, ...)$ be a uniform family of circuits, and let $f(n)$ and $g(n)$ be functions from the integers to the integers. We say that the parallel time of C is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. Furthermore, we say that the total work of C is at most $g(n)$ if for all $n \geq 0$ the size of C_n is at most $g(n)$. Finally, we let **PT/WK**$(f(n), g(n))$ be the class of all languages $L \subseteq \{0, 1\}^*$ such that there is a uniform family of circuits C deciding L with $O(f(n))$ parallel time and $O(g(n))$ work.

Models

Note

We have already seen different models of computation such as the Turing machine, the multistring variant, the RAM, and the nondeterministic Turing machine. For parallel computations, we will be using the Boolean circuit. This is because it has no "program counter," so its computational activity may take place at many gates concurrently.

Definition

Let C be a Boolean circuit where the size of C is the total number of gates in it, and the depth of C is the number of nodes in the longest path in C. Let $C = (C_0, C_1, \ldots)$ be a uniform family of circuits, and let $f(n)$ and $g(n)$ be functions from the integers to the integers. We say that the parallel time of C is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. Furthermore, we say that the total work of C is at most $g(n)$ if for all $n \geq 0$ the size of C_n is at most $g(n)$. Finally, we let **PT/WK**$(f(n), g(n))$ be the class of all languages $L \subseteq \{0,1\}^*$ such that there is a uniform family of circuits C deciding L with $O(f(n))$ parallel time and $O(g(n))$ work.

Parallel Random Access Machines

Definition

Recall that a RAM program is a sequence $\Pi = (\pi_1, ..., \pi_m)$ of instructions such as READ, ADD, LOAD, JUMP, etc. Also recall that we have a set of input registers. A PRAM program (parallel random access machine) is a sequence of RAM programs, $P = (\Pi_1, \Pi_2, ..., \Pi_q)$, one for each of q RAMS. Each machine can act independently of the others, but they all share the same input registers. q is not a constant but a function $q(m, n)$ where m is the number of integers in the input, and n is the total length of these integers. In other words, for each m and n, we have a different PRAM program $P_{m,n}$.

Parallel Random Access Machines

Definition

Recall that a RAM program is a sequence $\Pi = (\pi_1, ..., \pi_m)$ of instructions such as READ, ADD, LOAD, JUMP, etc. Also recall that we have a set of input registers. A PRAM program (parallel random access machine) is a sequence of RAM programs, $P = (\Pi_1, \Pi_2, ..., \Pi_q)$, one for each of q RAMS. Each machine can act independently of the others, but they all share the same input registers. q is not a constant but a function $q(m, n)$ where m is the number of integers in the input, and n is the total length of these integers. In other words, for each m and n, we have a different PRAM program $P_{m,n}$.

Parallel Random Access Machines

Definition

Recall that a RAM program is a sequence $\Pi = (\pi_1, ..., \pi_m)$ of instructions such as READ, ADD, LOAD, JUMP, etc. Also recall that we have a set of input registers. A PRAM program (parallel random access machine) is a sequence of RAM programs, $P = (\Pi_1, \Pi_2, ..., \Pi_q)$, one for each of q RAMS. Each machine can act independently of the others, but they all share the same input registers. q is not a constant but a function $q(m, n)$ where m is the number of integers in the input, and n is the total length of these integers. In other words, for each m and n, we have a different PRAM program $P_{m,n}$.

Parallel Random Access Machines

Definition

Recall that a RAM program is a sequence $\Pi = (\pi_1, ..., \pi_m)$ of instructions such as READ, ADD, LOAD, JUMP, etc. Also recall that we have a set of input registers. A PRAM program (parallel random access machine) is a sequence of RAM programs, $P = (\Pi_1, \Pi_2, ..., \Pi_q)$, one for each of q RAMS. Each machine can act independently of the others, but they all share the same input registers. q is not a constant but a function $q(m, n)$ where m is the number of integers in the input, and n is the total length of these integers. In other words, for each m and n, we have a different PRAM program $P_{m,n}$.

Parallel Random Access Machines

Definition

Recall that a RAM program is a sequence $\Pi = (\pi_1, ..., \pi_m)$ of instructions such as READ, ADD, LOAD, JUMP, etc. Also recall that we have a set of input registers. A PRAM program (parallel random access machine) is a sequence of RAM programs, $P = (\Pi_1, \Pi_2, ..., \Pi_q)$, one for each of q RAMS. Each machine can act independently of the others, but they all share the same input registers. q is not a constant but a function $q(m, n)$ where m is the number of integers in the input, and n is the total length of these integers. In other words, for each m and n, we have a different PRAM program $P_{m,n}$.

Parallel Random Access Machines

Definition

Recall that a RAM program is a sequence $\Pi = (\pi_1, ..., \pi_m)$ of instructions such as READ, ADD, LOAD, JUMP, etc. Also recall that we have a set of input registers. A PRAM program (parallel random access machine) is a sequence of RAM programs, $P = (\Pi_1, \Pi_2, ..., \Pi_q)$, one for each of q RAMS. Each machine can act independently of the others, but they all share the same input registers. q is not a constant but a function $q(m, n)$ where m is the number of integers in the input, and n is the total length of these integers. In other words, for each m and n, we have a different PRAM program $P_{m,n}$.

Optimal and Efficient Work

Note

We have previously mentioned how some algorithms require a specific amount of work. For example, we know that Matrix multiplication takes $\log n$ parallel time and n^3 work. Although the work done is efficient with respect to our $O(n^3)$ sequential algorithm, is the amount of work done optimal?

Definition

A parallel algorithm is said to be optimal if it involves the same amount of work as performed by the best sequential algorithm.

Note

*Our parallel algorithm is **not** optimal! There are other known algorithms such as Strassen's algorithm that has a better running time (approximately $O(n^{2.807})$), which means parallelizing these algorithms would be more better.*

Optimal and Efficient Work

Note

We have previously mentioned how some algorithms require a specific amount of work. For example, we know that Matrix multiplication takes $\log n$ parallel time and n^3 work. Although the work done is efficient with respect to our $O(n^3)$ sequential algorithm, is the amount of work done optimal?

Definition

A parallel algorithm is said to be optimal if it involves the same amount of work as performed by the best sequential algorithm.

Note

*Our parallel algorithm is **not** optimal! There are other known algorithms such as Strassen's algorithm that has a better running time (approximately $O(n^{2.807})$), which means parallelizing these algorithms would be more better.*

Optimal and Efficient Work

Note

We have previously mentioned how some algorithms require a specific amount of work. For example, we know that Matrix multiplication takes $\log n$ parallel time and n^3 work. Although the work done is efficient with respect to our $O(n^3)$ sequential algorithm, is the amount of work done optimal?

Definition

A parallel algorithm is said to be optimal if it involves the same amount of work as performed by the best sequential algorithm.

Note

*Our parallel algorithm is **not** optimal! There are other known algorithms such as Strassen's algorithm that has a better running time (approximately $O(n^{2.807})$), which means parallelizing these algorithms would be more better.*

Optimal and Efficient Work

Note

We have previously mentioned how some algorithms require a specific amount of work. For example, we know that Matrix multiplication takes $\log n$ *parallel time and* n^3 *work. Although the work done is efficient with respect to our* $O(n^3)$ *sequential algorithm, is the amount of work done optimal?*

Definition

A parallel algorithm is said to be optimal if it involves the same amount of work as performed by the best sequential algorithm.

Note

Our parallel algorithm is ***not*** *optimal! There are other known algorithms such as Strassen's algorithm that has a better running time (approximately* $O(n^{2.807})$*), which means parallelizing these algorithms would be more better.*

Optimal and Efficient Work

Note

We have previously mentioned how some algorithms require a specific amount of work. For example, we know that Matrix multiplication takes $\log n$ parallel time and n^3 work. Although the work done is efficient with respect to our $O(n^3)$ sequential algorithm, is the amount of work done optimal?

Definition

A parallel algorithm is said to be optimal if it involves the same amount of work as performed by the best sequential algorithm.

Note

*Our parallel algorithm is **not** optimal! There are other known algorithms such as Strassen's algorithm that has a better running time (approximately $O(n^{2.807})$), which means parallelizing these algorithms would be more better.*

Optimal and Efficient Work

Note

We have previously mentioned how some algorithms require a specific amount of work. For example, we know that Matrix multiplication takes $\log n$ parallel time and n^3 work. Although the work done is efficient with respect to our $O(n^3)$ sequential algorithm, is the amount of work done optimal?

Definition

A parallel algorithm is said to be optimal if it involves the same amount of work as performed by the best sequential algorithm.

Note

*Our parallel algorithm is **not** optimal! There are other known algorithms such as Strassen's algorithm that has a better running time (approximately $O(n^{2.807})$), which means parallelizing these algorithms would be more better.*

Class NC

Definition

We let **NC** = **PT/WK**($\log^k n, n^k$) be the class of all problems solvable in polylogarithmic parallel time with polynomial amount of total work.

Note

Although it is argued that ***NC*** *captures the notion of "problems satisfactorily solved by parallel computers" much like* ***P*** *captures the notion of efficient computability in the sequential context, the argument is not as convincing. This is because in sequential computation, the difference between polynomial and exponential (such as 2^n and n^3) is real and dramatic for when n is small. Although $\log^3 n$ is smaller than $\sqrt{n}$, we do not see the difference until $n = 10^{12}$, and the notion of "polynomial number of processors" is absurd.*

Class NC

Definition

We let **NC** = **PT**/**WK**($\log^k n, n^k$) be the class of all problems solvable in polylogarithmic parallel time with polynomial amount of total work.

Note

*Although it is argued that **NC** captures the notion of "problems satisfactorily solved by parallel computers" much like **P** captures the notion of efficient computability in the sequential context, the argument is not as convincing. This is because in sequential computation, the difference between polynomial and exponential (such as 2^n and n^3) is real and dramatic for when n is small. Although $\log^3 n$ is smaller than $\sqrt{n}$, we do not see the difference until $n = 10^{12}$, and the notion of "polynomial number of processors" is absurd.*

Class NC

Definition

We let **NC** = **PT/WK**($\log^k n, n^k$) be the class of all problems solvable in polylogarithmic parallel time with polynomial amount of total work.

Note

*Although it is argued that **NC** captures the notion of "problems satisfactorily solved by parallel computers" much like **P** captures the notion of efficient computability in the sequential context, the argument is not as convincing. This is because in sequential computation, the difference between polynomial and exponential (such as 2^n and n^3) is real and dramatic for when n is small. Although $\log^3 n$ is smaller than $\sqrt{n}$, we do not see the difference until $n = 10^{12}$, and the notion of "polynomial number of processors" is absurd.*

Class NC

Definition

We let **NC** = **PT**/**WK**($\log^k n, n^k$) be the class of all problems solvable in polylogarithmic parallel time with polynomial amount of total work.

Note

*Although it is argued that **NC** captures the notion of ”problems satisfactorily solved by parallel computers” much like **P** captures the notion of efficient computability in the sequential context, the argument is not as convincing. This is because in sequential computation, the difference between polynomial and exponential (such as 2^n and n^3) is real and dramatic for when n is small. Although $\log^3 n$ is smaller than $\sqrt{n}$, we do not see the difference until $n = 10^{12}$, and the notion of ”polynomial number of processors” is absurd.*

NC Refined

Definition

We let $\mathbf{NC}_j = \mathbf{PT/WK}(\log^j n, n^k)$ be the subset of **NC** in which the parallel time is $O(\log^j n)$; the free parameter k means that we allow any degree in the polynomial accounting for the total work.

Example

Reachability would be $\mathbf{NC}_2$ since it can be computed in $O(\log^2 n)$ parallel time.

NC Refined

Definition

We let $\mathbf{NC}_j = \mathbf{PT/WK}(\log^j n, n^k)$ be the subset of **NC** in which the parallel time is $O(\log^j n)$; the free parameter k means that we allow any degree in the polynomial accounting for the total work.

Example

REACHABILITY would be $\mathbf{NC}_2$ since it can be computed in $O(\log^2 n)$ parallel time.

NC and P

Note

Since the amount of work involved in solving any problem in ***NC*** *is bounded by a polynomial, we can see that* ***NC*** $\subseteq$ ***P****. But is* ***NC*** $=$ ***P****? This open problem is the counterpart of the* ***P*** $=$ ***NP*** *for parallel computations. This is most likely not true since if* ***NC*** $=$ ***P****, then we are saying that any polynomial-time solvable problem could be parallelized.*

NC and P

Note

Since the amount of work involved in solving any problem in **NC** *is bounded by a polynomial, we can see that* **NC** $\subseteq$ **P**. *But is* **NC** $=$ **P**? *This open problem is the counterpart of the* **P** $=$ **NP** *for parallel computations. This is most likely not true since if* **NC** $=$ **P**, *then we are saying that any polynomial-time solvable problem could be parallelized.*

NC and P

Note

Since the amount of work involved in solving any problem in **NC** *is bounded by a polynomial, we can see that* **NC** $\subseteq$ **P**. *But is* **NC** $=$ **P**? *This open problem is the counterpart of the* **P** $=$ **NP** *for parallel computations. This is most likely not true since if* **NC** $=$ **P**, *then we are saying that any polynomial-time solvable problem could be parallelized.*

NC and P

Note

*Since the amount of work involved in solving any problem in **NC** is bounded by a polynomial, we can see that **NC** $\subseteq$ **P**. But is **NC** $=$ **P**? This open problem is the counterpart of the **P** $=$ **NP** for parallel computations. This is most likely not true since if **NC** $=$ **P**, then we are saying that any polynomial-time solvable problem could be parallelized.*

Outline

P-completeness

Definition

A decision problem is **P**-complete if it is in **P** and that every problem in **P** can be reduced to it by using an appropriate reduction.

Note

***P**-complete problems are the least likely to be in **NC**. However, we must first show that our logarithmic-space reductions preserve parallel complexity.*

Theorem

*If $L \in$ **NC** reduces to L', then $L' \in$ **NC**.*

P-completeness

Definition

A decision problem is **P**-complete if it is in **P** and that every problem in **P** can be reduced to it by using an appropriate reduction.

Note

***P**-complete problems are the least likely to be in **NC**. However, we must first show that our logarithmic-space reductions preserve parallel complexity.*

Theorem

*If $L \in$ **NC** reduces to L', then $L' \in$ **NC**.*

P-completeness

Definition

A decision problem is **P**-complete if it is in **P** and that every problem in **P** can be reduced to it by using an appropriate reduction.

Note

***P**-complete problems are the least likely to be in **NC**. However, we must first show that our logarithmic-space reductions preserve parallel complexity.*

Theorem

*If $L \in$ **NC** reduces to L', then $L' \in$ **NC**.*

P-completeness

Definition

A decision problem is **P**-complete if it is in **P** and that every problem in **P** can be reduced to it by using an appropriate reduction.

Note

***P**-complete problems are the least likely to be in **NC**. However, we must first show that our logarithmic-space reductions preserve parallel complexity.*

Theorem

*If $L \in$ **NC** reduces to L', then $L' \in$ **NC**.*

P-completeness (Contd.)

Proof.

Let R be the logarithmic-space reduction from L to L'. There does exist a logarithmic space-bounded Turing machine, which we will call R', that accepts the input (x, i) (where x is the input string and i is the binary representation of an integer no larger than $|R(x)|$) if and only if the ith bit of $R(x)$ is one. We use this setup so we can solve the REACHABILITY problem for R' on input (x, i) to compute the ith bit of $R(x)$. Therefore, if we solve these problems in parallel by $\mathbf{NC}_2$ circuits, we can compute all bits of $R(x)$. Once we have $R(x)$ we can use the **NC** circuit for L' to tell whether $x \in L$, all in **NC**. $\square$

Corollary

If $L \in \mathbf{NC}_j$ reduces to L', where $j \geq 2$, then $L' \in \mathbf{NC}_j$.

P-completeness (Contd.)

Proof.

Let R be the logarithmic-space reduction from L to L'. There does exist a logarithmic space-bounded Turing machine, which we will call R', that accepts the input (x, i) (where x is the input string and i is the binary representation of an integer no larger than $|R(x)|$) if and only if the ith bit of $R(x)$ is one. We use this setup so we can solve the REACHABILITY problem for R' on input (x, i) to compute the ith bit of $R(x)$. Therefore, if we solve these problems in parallel by $\mathbf{NC}_2$ circuits, we can compute all bits of $R(x)$. Once we have $R(x)$ we can use the **NC** circuit for L' to tell whether $x \in L$, all in **NC**. □

Corollary

If $L \in \mathbf{NC}_j$ reduces to L', where $j \geq 2$, then $L' \in \mathbf{NC}_j$.

P-completeness (Contd.)

Proof.

Let R be the logarithmic-space reduction from L to L'. There does exist a logarithmic space-bounded Turing machine, which we will call R', that accepts the input (x, i) (where x is the input string and i is the binary representation of an integer no larger than $|R(x)|$) if and only if the ith bit of $R(x)$ is one. We use this setup so we can solve the REACHABILITY problem for R' on input (x, i) to compute the ith bit of $R(x)$. Therefore, if we solve these problems in parallel by $\mathbf{NC}_2$ circuits, we can compute all bits of $R(x)$. Once we have $R(x)$ we can use the **NC** circuit for L' to tell whether $x \in L$, all in **NC**. $\square$

Corollary

If $L \in \mathbf{NC}_j$ reduces to L', where $j \geq 2$, then $L' \in \mathbf{NC}_j$.

P-completeness (Contd.)

Proof.

Let R be the logarithmic-space reduction from L to L'. There does exist a logarithmic space-bounded Turing machine, which we will call R', that accepts the input (x, i) (where x is the input string and i is the binary representation of an integer no larger than $|R(x)|$) if and only if the ith bit of $R(x)$ is one. We use this setup so we can solve the REACHABILITY problem for R' on input (x, i) to compute the ith bit of $R(x)$. Therefore, if we solve these problems in parallel by $\mathbf{NC}_2$ circuits, we can compute all bits of $R(x)$. Once we have $R(x)$ we can use the **NC** circuit for L' to tell whether $x \in L$, all in **NC**. □

Corollary

If $L \in \mathbf{NC}_j$ reduces to L', where $j \geq 2$, then $L' \in \mathbf{NC}_j$.

P-completeness (Contd.)

Proof.

Let R be the logarithmic-space reduction from L to L'. There does exist a logarithmic space-bounded Turing machine, which we will call R', that accepts the input (x, i) (where x is the input string and i is the binary representation of an integer no larger than $|R(x)|$) if and only if the ith bit of $R(x)$ is one. We use this setup so we can solve the REACHABILITY problem for R' on input (x, i) to compute the ith bit of $R(x)$. Therefore, if we solve these problems in parallel by $\mathbf{NC}_2$ circuits, we can compute all bits of $R(x)$. Once we have $R(x)$ we can use the **NC** circuit for L' to tell whether $x \in L$, all in **NC**. $\square$

Corollary

If $L \in \mathbf{NC}_j$ reduces to L', where $j \geq 2$, then $L' \in \mathbf{NC}_j$.

P-completeness (Contd.)

Proof.

Let R be the logarithmic-space reduction from L to L'. There does exist a logarithmic space-bounded Turing machine, which we will call R', that accepts the input (x, i) (where x is the input string and i is the binary representation of an integer no larger than $|R(x)|$) if and only if the ith bit of $R(x)$ is one. We use this setup so we can solve the REACHABILITY problem for R' on input (x, i) to compute the ith bit of $R(x)$. Therefore, if we solve these problems in parallel by $\mathbf{NC}_2$ circuits, we can compute all bits of $R(x)$. Once we have $R(x)$ we can use the **NC** circuit for L' to tell whether $x \in L$, all in **NC**. □

Corollary

If $L \in \mathbf{NC}_j$ reduces to L', where $j \geq 2$, then $L' \in \mathbf{NC}_j$.

Outline

Odd Max Flow

Problem

Given an network $N = (V, E, s, t, c)$, is the value of the maximum flow odd?

Theorem

Odd Max Flow *is* ***P****-complete.*

Proof

We already know that this problem is in **P** since we have an $O(n^5)$ algorithm by getting the maximum flow of the shortest path from s to t. To show completeness, we will reduce Monotone Circuit Value to the Odd Max Flow problem. Recall that the Monotone Circuit Value problem states that given a set of gates $g_1, \ldots, g_n$ where each gate is either an AND gate, an OR gate, or a constant value that is true or false, we wish to compute the value of g_n.

ODD MAX FLOW

Problem

Given an network $N = (V, E, s, t, c)$, is the value of the maximum flow odd?

Theorem

ODD MAX FLOW *is **P**-complete.*

Proof

We already know that this problem is in **P** since we have an $O(n^5)$ algorithm by getting the maximum flow of the shortest path from s to t. To show completeness, we will reduce MONOTONE CIRCUIT VALUE to the ODD MAX FLOW problem. Recall that the MONOTONE CIRCUIT VALUE problem states that given a set of gates $g_1, \dots, g_n$ where each gate is either an AND gate, an OR gate, or a constant value that is true or false, we wish to compute the value of g_n.

Odd Max Flow

Problem

Given an network $N = (V, E, s, t, c)$, is the value of the maximum flow odd?

Theorem

ODD MAX FLOW *is* ***P****-complete.*

Proof

We already know that this problem is in **P** since we have an $O(n^5)$ algorithm by getting the maximum flow of the shortest path from s to t. To show completeness, we will reduce MONOTONE CIRCUIT VALUE to the ODD MAX FLOW problem. Recall that the MONOTONE CIRCUIT VALUE problem states that given a set of gates $g_1, \ldots, g_n$ where each gate is either an AND gate, an OR gate, or a constant value that is true or false, we wish to compute the value of g_n.

ODD MAX FLOW

Problem

Given an network $N = (V, E, s, t, c)$, is the value of the maximum flow odd?

Theorem

ODD MAX FLOW *is* ***P****-complete.*

Proof

We already know that this problem is in **P** since we have an $O(n^5)$ algorithm by getting the maximum flow of the shortest path from s to t. To show completeness, we will reduce MONOTONE CIRCUIT VALUE to the ODD MAX FLOW problem. Recall that the MONOTONE CIRCUIT VALUE problem states that given a set of gates $g_1, \ldots, g_n$ where each gate is either an AND gate, an OR gate, or a constant value that is true or false, we wish to compute the value of g_n.

Odd Max Flow

Problem

Given an network $N = (V, E, s, t, c)$, is the value of the maximum flow odd?

Theorem

Odd Max Flow *is **P**-complete.*

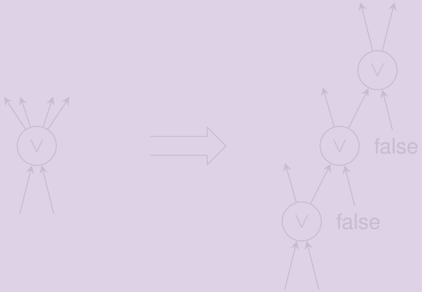
Proof

We already know that this problem is in **P** since we have an $O(n^5)$ algorithm by getting the maximum flow of the shortest path from s to t. To show completeness, we will reduce Monotone Circuit Value to the Odd Max Flow problem. Recall that the Monotone Circuit Value problem states that given a set of gates $g_1, ..., g_n$ where each gate is either an AND gate, an OR gate, or a constant value that is true or false, we wish to compute the value of g_n.

Odd Max Flow (Contd.)

Proof

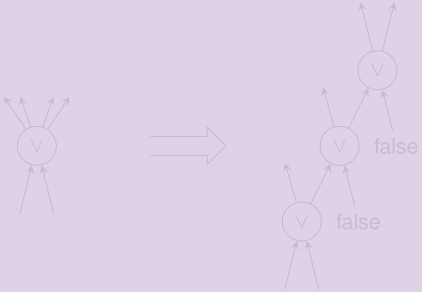
Given a monotone circuit C, assume that the output gate of C is an OR gate, and no gate of C has outdegree more than two. We can modify C for this restriction by adding additional OR gates to any gate whose outdegree is more than two where the inputs of these gates are **false**.



Odd Max Flow (Contd.)

Proof

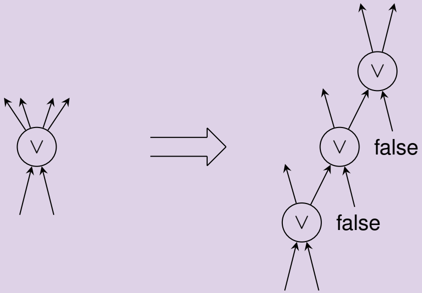
Given a monotone circuit C, assume that the output gate of C is an OR gate, and no gate of C has outdegree more than two. We can modify C for this restriction by adding additional OR gates to any gate whose outdegree is more than two where the inputs of these gates are **false**.



Odd Max Flow (Contd.)

Proof

Given a monotone circuit C, assume that the output gate of C is an OR gate, and no gate of C has outdegree more than two. We can modify C for this restriction by adding additional OR gates to any gate whose outdegree is more than two where the inputs of these gates are **false**.



Odd Max Flow (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

ODD MAX FLOW (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

Odd Max Flow (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

ODD MAX FLOW (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

ODD MAX FLOW (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

Odd Max Flow (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

ODD MAX FLOW (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

ODD MAX FLOW (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

Odd Max Flow (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

Odd Max Flow (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

Odd Max Flow (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

Odd Max Flow (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

ODD MAX FLOW (Contd.)

Labeling

We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

ODD MAX FLOW (Contd.)

Labeling

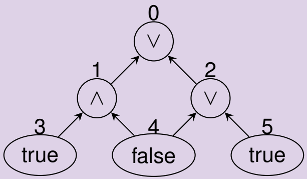
We also assume that the gates have been assigned consecutive numbers such that each gate has smaller label than is predecessor. This will mean that the output gate will have the label 0.

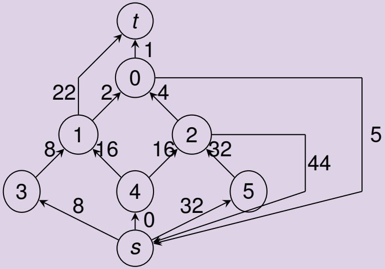
Construction

The construction is as follows: let each node in the network N produced from C be a gate plus the nodes s and t (the source and sink). For each edge outgoing from s that connects to a **true** gate i, let the capacity of the edge be $d2^i$, where i is the label and d is the outdegree of gate i. If we connect to a **false** gate, we let the capacity be 0. From a **true** or **false** gate i to another gate, the capacity of the edge is 2^i. From an OR or AND gate i to another gate, the capacity of the edge is also 2^i. From the output gate to edge t, there is an edge of capacity one.
We now consider any AND or OR gate i. We know that it has several incoming and at most two outgoing edges. Since the capacity of the outgoing edges is 2^i and the capacities of the incoming edges are at least twice that, we have a surplus of incoming capacity, denoted at $S(i)$. If i is an AND gate, we make an edge from i to t with capacity $S(i)$. If i is an OR gate, we make an edge from i to s with capacity $S(i)$.

Odd Max Flow (Contd.)

Proof

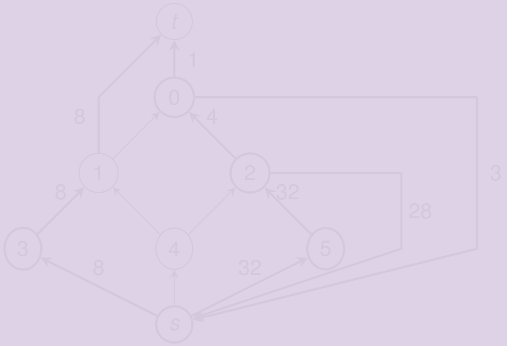




ODD MAX FLOW (Contd.)

Proof

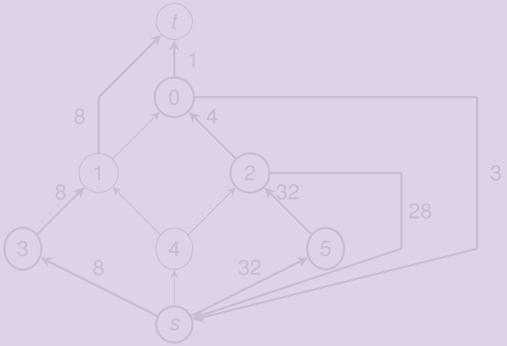
Given a flow f, we say that a gate is full if all of its outgoing edges are filled to capacity. A gate is empty if its outgoing edges have zero flow. We say that f is standard if all **true** gates are full and all **false** gates are empty.



ODD MAX FLOW (Contd.)

Proof

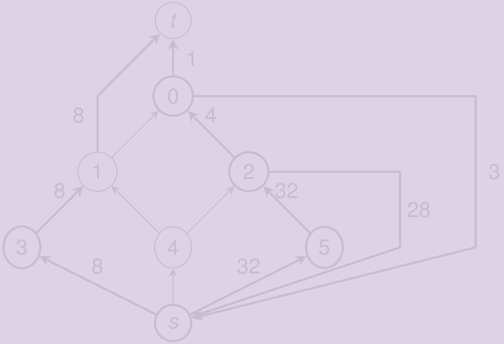
Given a flow f, we say that a gate is full if all of its outgoing edges are filled to capacity. A gate is empty if its outgoing edges have zero flow. We say that f is standard if all **true** gates are full and all **false** gates are empty.



ODD MAX FLOW (Contd.)

Proof

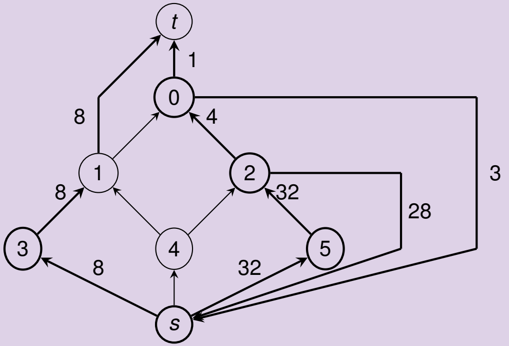
Given a flow *f*, we say that a gate is full if all of its outgoing edges are filled to capacity. A gate is empty if its outgoing edges have zero flow. We say that *f* is standard if all **true** gates are full and all **false** gates are empty.



Odd Max Flow (Contd.)

Proof

Given a flow *f*, we say that a gate is full if all of its outgoing edges are filled to capacity. A gate is empty if its outgoing edges have zero flow. We say that *f* is standard if all **true** gates are full and all **false** gates are empty.



Odd Max Flow (Contd.)

Proof

We will show that a standard flow always exists and that it is the maximum flow. We start by pushing the maximum flow to each input gate outgoing from s that is **true**. This means that each **true** input gate will be full, and each **false** input gate will be empty. By induction, all OR gates that are **true** will have at least one incoming edge with the flow at maximum capacity, so there will be enough to go out and possibly have a surplus. If the OR gate if **false**, then there is no incoming flow. If an AND gate is **true**, then both incoming edges are at maximum capacity, so there will be enough outgoing flow and possibly a surplus. If an AND gate is **false**, then there is at most one incoming edge with the flow at capacity, which can be directed to the surplus edge.

Odd Max Flow (Contd.)

Proof

We will show that a standard flow always exists and that it is the maximum flow. We start by pushing the maximum flow to each input gate outgoing from s that is **true**. This means that each **true** input gate will be full, and each **false** input gate will be empty. By induction, all OR gates that are **true** will have at least one incoming edge with the flow at maximum capacity, so there will be enough to go out and possibly have a surplus. If the OR gate if **false**, then there is no incoming flow. If an AND gate is **true**, then both incoming edges are at maximum capacity, so there will be enough outgoing flow and possibly a surplus. If an AND gate is **false**, then there is at most one incoming edge with the flow at capacity, which can be directed to the surplus edge.

ODD MAX FLOW (Contd.)

Proof

We will show that a standard flow always exists and that it is the maximum flow. We start by pushing the maximum flow to each input gate outgoing from s that is **true**. This means that each **true** input gate will be full, and each **false** input gate will be empty. By induction, all OR gates that are **true** will have at least one incoming edge with the flow at maximum capacity, so there will be enough to go out and possibly have a surplus. If the OR gate if **false**, then there is no incoming flow. If an AND gate is **true**, then both incoming edges are at maximum capacity, so there will be enough outgoing flow and possibly a surplus. If an AND gate is **false**, then there is at most one incoming edge with the flow at capacity, which can be directed to the surplus edge.

Odd Max Flow (Contd.)

Proof

We will show that a standard flow always exists and that it is the maximum flow. We start by pushing the maximum flow to each input gate outgoing from s that is **true**. This means that each **true** input gate will be full, and each **false** input gate will be empty. By induction, all OR gates that are **true** will have at least one incoming edge with the flow at maximum capacity, so there will be enough to go out and possibly have a surplus. If the OR gate if **false**, then there is no incoming flow. If an AND gate is **true**, then both incoming edges are at maximum capacity, so there will be enough outgoing flow and possibly a surplus. If an AND gate is **false**, then there is at most one incoming edge with the flow at capacity, which can be directed to the surplus edge.

Odd Max Flow (Contd.)

Proof

We will show that a standard flow always exists and that it is the maximum flow. We start by pushing the maximum flow to each input gate outgoing from *s* that is **true**. This means that each **true** input gate will be full, and each **false** input gate will be empty. By induction, all OR gates that are **true** will have at least one incoming edge with the flow at maximum capacity, so there will be enough to go out and possibly have a surplus. If the OR gate if **false**, then there is no incoming flow. If an AND gate is **true**, then both incoming edges are at maximum capacity, so there will be enough outgoing flow and possibly a surplus. If an AND gate is **false**, then there is at most one incoming edge with the flow at capacity, which can be directed to the surplus edge.

Odd Max Flow (Contd.)

Proof

We will show that a standard flow always exists and that it is the maximum flow. We start by pushing the maximum flow to each input gate outgoing from *s* that is **true**. This means that each **true** input gate will be full, and each **false** input gate will be empty. By induction, all OR gates that are **true** will have at least one incoming edge with the flow at maximum capacity, so there will be enough to go out and possibly have a surplus. If the OR gate if **false**, then there is no incoming flow. If an AND gate is **true**, then both incoming edges are at maximum capacity, so there will be enough outgoing flow and possibly a surplus. If an AND gate is **false**, then there is at most one incoming edge with the flow at capacity, which can be directed to the surplus edge.

ODD MAX FLOW (Contd.)

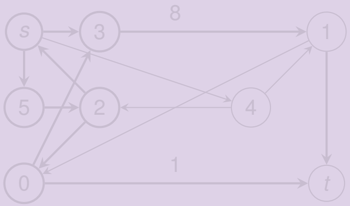
Proof

We will show that a standard flow always exists and that it is the maximum flow. We start by pushing the maximum flow to each input gate outgoing from s that is **true**. This means that each **true** input gate will be full, and each **false** input gate will be empty. By induction, all OR gates that are **true** will have at least one incoming edge with the flow at maximum capacity, so there will be enough to go out and possibly have a surplus. If the OR gate if **false**, then there is no incoming flow. If an AND gate is **true**, then both incoming edges are at maximum capacity, so there will be enough outgoing flow and possibly a surplus. If an AND gate is **false**, then there is at most one incoming edge with the flow at capacity, which can be directed to the surplus edge.

Odd Max Flow (Contd.)

Proof.

We now separate N into two groups: one group will contain s and all of the **true** gates, and the other group will contain t and all of the **false** gates.

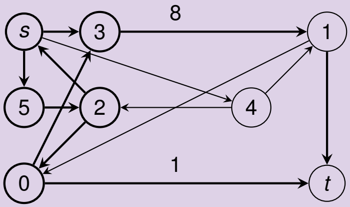


Note that there are two types edges going from the first group to the second: edges from true OR gates (or true input gates) to false AND gates, or from true AND gates (or output gate if true) to t. Both types are full and account for all flow going into t. Therefore the capacity of this cut is the value of f and is the maximum by the max-flow min-cut theorem. Finally, notice that all flows are even integers except possibly from the output gate to t. This means that the value of the max flow is odd if and only if the output gate if full, which happens if and only if the output gate is **true**. □

ODD MAX FLOW (Contd.)

Proof.

We now separate N into two groups: one group will contain s and all of the **true** gates, and the other group will contain t and all of the **false** gates.

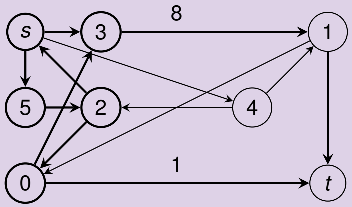


Note that there are two types edges going from the first group to the second: edges from true OR gates (or true input gates) to false AND gates, or from true AND gates (or output gate if true) to t. Both types are full and account for all flow going into t.
Therefore the capacity of this cut is the value of f and is the maximum by the max-flow min-cut theorem. Finally, notice that all flows are even integers except possibly from the output gate to t. This means that the value of the max flow is odd if and only if the output gate if full, which happens if and only if the output gate is **true**.

ODD MAX FLOW (Contd.)

Proof.

We now separate N into two groups: one group will contain s and all of the **true** gates, and the other group will contain t and all of the **false** gates.

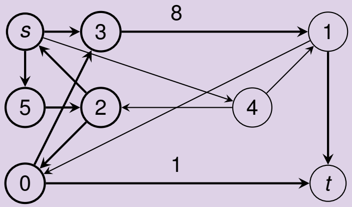


Note that there are two types edges going from the first group to the second: edges from true OR gates (or true input gates) to false AND gates, or from true AND gates (or output gate if true) to t. Both types are full and account for all flow going into t. Therefore the capacity of this cut is the value of f and is the maximum by the max-flow min-cut theorem. Finally, notice that all flows are even integers except possibly from the output gate to t. This means that the value of the max flow is odd if and only if the output gate if full, which happens if and only if the output gate is **true**.

Odd Max Flow (Contd.)

Proof.

We now separate N into two groups: one group will contain s and all of the **true** gates, and the other group will contain t and all of the **false** gates.

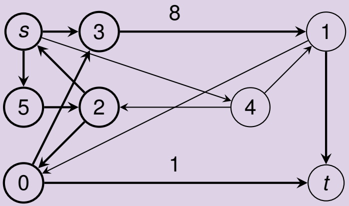


Note that there are two types edges going from the first group to the second: edges from true OR gates (or true input gates) to false AND gates, or from true AND gates (or output gate if true) to t. Both types are full and account for all flow going into t. Therefore the capacity of this cut is the value of f and is the maximum by the max-flow min-cut theorem. Finally, notice that all flows are even integers except possibly from the output gate to t. This means that the value of the max flow is odd if and only if the output gate if full, which happens if and only if the output gate is **true**. □

ODD MAX FLOW (Contd.)

Proof.

We now separate N into two groups: one group will contain s and all of the **true** gates, and the other group will contain t and all of the **false** gates.

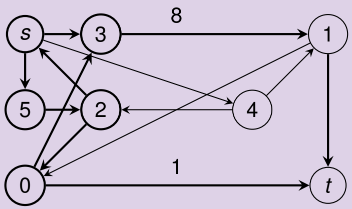


Note that there are two types edges going from the first group to the second: edges from true OR gates (or true input gates) to false AND gates, or from true AND gates (or output gate if true) to t. Both types are full and account for all flow going into t. Therefore the capacity of this cut is the value of f and is the maximum by the max-flow min-cut theorem. Finally, notice that all flows are even integers except possibly from the output gate to t. This means that the value of the max flow is odd if and only if the output gate if full, which happens if and only if the output gate is **true**. □

Odd Max Flow (Contd.)

Proof.

We now separate *N* into two groups: one group will contain *s* and all of the **true** gates, and the other group will contain *t* and all of the **false** gates.

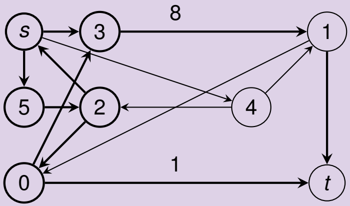


Note that there are two types edges going from the first group to the second: edges from true OR gates (or true input gates) to false AND gates, or from true AND gates (or output gate if true) to *t*. Both types are full and account for all flow going into *t*. Therefore the capacity of this cut is the value of *f* and is the maximum by the max-flow min-cut theorem. Finally, notice that all flows are even integers except possibly from the output gate to *t*. This means that the value of the max flow is odd if and only if the output gate if full, which happens if and only if the output gate is **true**. □

Odd Max Flow (Contd.)

Proof.

We now separate N into two groups: one group will contain s and all of the **true** gates, and the other group will contain t and all of the **false** gates.



Note that there are two types edges going from the first group to the second: edges from true OR gates (or true input gates) to false AND gates, or from true AND gates (or output gate if true) to t. Both types are full and account for all flow going into t. Therefore the capacity of this cut is the value of f and is the maximum by the max-flow min-cut theorem. Finally, notice that all flows are even integers except possibly from the output gate to t. This means that the value of the max flow is odd if and only if the output gate if full, which happens if and only if the output gate is **true**. □

The Class RNC

Definition

The class **RNC** consists of all languages L that have a randomized algorithm that is solvable in polylogarithmic parallel time with polynomial amount of total work, and the probability of producing a correct solution is at least $\frac{1}{2}$.

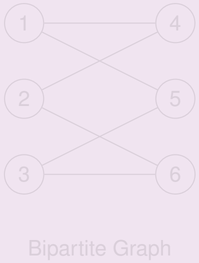
Outline

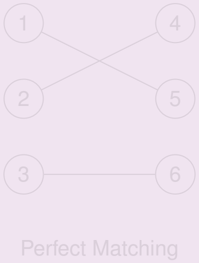
Perfect Matching is in RNC

Problem

Given a bipartite graph, does it have a perfect matching, one where each node is matched to exactly one other node and no two matchings share the same node?

Example



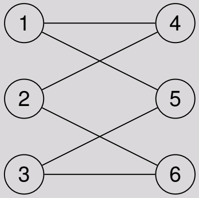


Perfect Matching is in RNC

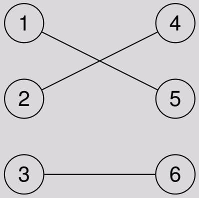
Problem

Given a bipartite graph, does it have a perfect matching, one where each node is matched to exactly one other node and no two matchings share the same node?

Example



Bipartite Graph



Perfect Matching

Perfect Matching is in RNC (Contd.)

Note

We can use the minimum-weight perfect matching problem to show this. Suppose that each edge $(u_i, u_j) \in E$ has a weight w_{ij} associated with it, and we want not just any perfect matching, but the matching π that minimizes $w(\pi) = \sum_{i=1}^{n} w_{i,\pi(i)}$. There is an ***NC*** *algorithm for this when the weights are small and polynomial and the minimum-weight matching is unique.*

Perfect Matching is in RNC (Contd.)

Note

We can use the minimum-weight perfect matching problem to show this. Suppose that each edge $(u_i, u_j) \in E$ *has a weight* w_{ij} *associated with it, and we want not just any perfect matching, but the matching* π *that minimizes* $w(\pi) = \sum_{i=1}^{n} w_{i,\pi(i)}$. *There is an **NC** algorithm for this when the weights are small and polynomial and the minimum-weight matching is unique.*

Perfect Matching is in RNC (Contd.)

Note

We can use the minimum-weight perfect matching problem to show this. Suppose that each edge $(u_i, u_j) \in E$ *has a weight* w_{ij} *associated with it, and we want not just any perfect matching, but the matching* π *that minimizes* $w(\pi) = \sum_{i=1}^{n} w_{i,\pi(i)}$. *There is an* ***NC*** *algorithm for this when the weights are small and polynomial and the minimum-weight matching is unique.*

Perfect Matching is in RNC (Contd.)

Algorithm

We define a matrix $A^{G,w}$ whose i, jth elements is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. Recall that $\det A^{G,w} = \sum_{\pi} \sigma(\pi) \Pi_{i=1}^{n} A_{i,\pi(i)}^{G,w}$, which is actually $2^{w(\pi)}$ since we are not concerned about permutations that are not perfect matchings (those terms are zero). Since the minimum weight is unique, let this be w^*. This means all terms of $\det A^{G,w}$ are multiples of 2^{w^*}, and all of them but one will be even multiples of 2^{w^*}. Therefore, 2^{w^*} is the highest power of two that divides $\det A^{G,w}$, which means we can calculate w^* efficiently in parallel.
We first get $\det A^{G,w}$ using our **NC** algorithm for determinants. w^* will be the number of trailing zeros in the binary representation of the determinant. To see whether an edge (u_i, v_j) is in the minimum-weight perfect matching, we set the weight of this edge to 0. If the new minimum weight is $w^* - w_{ij}$, then our edge is in this matching. Each of these tests can be done in parallel.

Perfect Matching is in RNC (Contd.)

Algorithm

We define a matrix $A^{G,w}$ whose i, jth elements is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. Recall that $\det A^{G,w} = \sum_{\pi} \sigma(\pi) \Pi_{i=1}^{n} A^{G,w}_{i,\pi(i)}$, which is actually $2^{w(\pi)}$ since we are not concerned about permutations that are not perfect matchings (those terms are zero). Since the minimum weight is unique, let this be w^*. This means all terms of $\det A^{G,w}$ are multiples of 2^{w^*}, and all of them but one will be even multiples of 2^{w^*}. Therefore, 2^{w^*} is the highest power of two that divides $\det A^{G,w}$, which means we can calculate w^* efficiently in parallel.

We first get $\det A^{G,w}$ using our **NC** algorithm for determinants. w^* will be the number of trailing zeros in the binary representation of the determinant. To see whether an edge (u_i, v_j) is in the minimum-weight perfect matching, we set the weight of this edge to 0. If the new minimum weight is $w^* - w_{ij}$, then our edge is in this matching. Each of these tests can be done in parallel.

Perfect Matching is in RNC (Contd.)

Algorithm

We define a matrix $A^{G,w}$ whose i, jth elements is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. Recall that $\det A^{G,w} = \sum_{\pi} \sigma(\pi) \Pi_{i=1}^{n} A_{i,\pi(i)}^{G,w}$, which is actually $2^{w(\pi)}$ since we are not concerned about permutations that are not perfect matchings (those terms are zero). Since the minimum weight is unique, let this be $w*$. This means all terms of $\det A^{G,w}$ are multiples of 2^{w^*}, and all of them but one will be even multiples of 2^{w^*}. Therefore, 2^{w^*} is the highest power of two that divides $\det A^{G,w}$, which means we can calculate $w*$ efficiently in parallel.

We first get $\det A^{G,w}$ using our **NC** algorithm for determinants. w^* will be the number of trailing zeros in the binary representation of the determinant. To see whether an edge (u_i, v_j) is in the minimum-weight perfect matching, we set the weight of this edge to 0. If the new minimum weight is $w^* - w_{ij}$, then our edge is in this matching. Each of these tests can be done in parallel.

Perfect Matching is in RNC (Contd.)

Algorithm

We define a matrix $A^{G,w}$ whose i, jth elements is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. Recall that $\det A^{G,w} = \sum_{\pi} \sigma(\pi)\Pi_{i=1}^{n} A_{i,\pi(i)}^{G,w}$, which is actually $2^{w(\pi)}$ since we are not concerned about permutations that are not perfect matchings (those terms are zero). Since the minimum weight is unique, let this be w^*. This means all terms of $\det A^{G,w}$ are multiples of 2^{w^*}, and all of them but one will be even multiples of 2^{w^*}. Therefore, 2^{w^*} is the highest power of two that divides $\det A^{G,w}$, which means we can calculate w^* efficiently in parallel.
We first get $\det A^{G,w}$ using our **NC** algorithm for determinants. w^* will be the number of trailing zeros in the binary representation of the determinant. To see whether an edge (u_i, v_j) is in the minimum-weight perfect matching, we set the weight of this edge to 0. If the new minimum weight is $w^* - w_{ij}$, then our edge is in this matching. Each of these tests can be done in parallel.

Perfect Matching is in RNC (Contd.)

Algorithm

We define a matrix $A^{G,w}$ whose i,jth elements is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. Recall that $\det A^{G,w} = \sum_{\pi} \sigma(\pi)\Pi_{i=1}^{n} A^{G,w}_{i,\pi(i)}$, which is actually $2^{w(\pi)}$ since we are not concerned about permutations that are not perfect matchings (those terms are zero). Since the minimum weight is unique, let this be w^*. This means all terms of $\det A^{G,w}$ are multiples of 2^{w^*}, and all of them but one will be even multiples of 2^{w^*}. Therefore, 2^{w^*} is the highest power of two that divides $\det A^{G,w}$, which means we can calculate w^* efficiently in parallel.
We first get $\det A^{G,w}$ using our **NC** algorithm for determinants. w^* will be the number of trailing zeros in the binary representation of the determinant. To see whether an edge (u_i, v_j) is in the minimum-weight perfect matching, we set the weight of this edge to 0. If the new minimum weight is $w^* - w_{ij}$, then our edge is in this matching. Each of these tests can be done in parallel.

Perfect Matching is in RNC (Contd.)

Algorithm

We define a matrix $A^{G,w}$ whose i,jth elements is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. Recall that $\det A^{G,w} = \sum_{\pi} \sigma(\pi)\Pi_{i=1}^{n} A^{G,w}_{i,\pi(i)}$, which is actually $2^{w(\pi)}$ since we are not concerned about permutations that are not perfect matchings (those terms are zero). Since the minimum weight is unique, let this be w^*. This means all terms of $\det A^{G,w}$ are multiples of 2^{w^*}, and all of them but one will be even multiples of 2^{w^*}. Therefore, 2^{w^*} is the highest power of two that divides $\det A^{G,w}$, which means we can calculate w^* efficiently in parallel.
We first get $\det A^{G,w}$ using our **NC** algorithm for determinants. w^* will be the number of trailing zeros in the binary representation of the determinant. To see whether an edge (u_i, v_j) is in the minimum-weight perfect matching, we set the weight of this edge to 0. If the new minimum weight is $w^* - w_{ij}$, then our edge is in this matching. Each of these tests can be done in parallel.

Perfect Matching is in RNC (Contd.)

Algorithm

We define a matrix $A^{G,w}$ whose i, jth elements is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. Recall that $\det A^{G,w} = \sum_{\pi} \sigma(\pi)\Pi_{i=1}^{n} A^{G,w}_{i,\pi(i)}$, which is actually $2^{w(\pi)}$ since we are not concerned about permutations that are not perfect matchings (those terms are zero). Since the minimum weight is unique, let this be w^*. This means all terms of $\det A^{G,w}$ are multiples of 2^{w^*}, and all of them but one will be even multiples of 2^{w^*}. Therefore, 2^{w^*} is the highest power of two that divides $\det A^{G,w}$, which means we can calculate w^* efficiently in parallel.
We first get $\det A^{G,w}$ using our **NC** algorithm for determinants. w^* will be the number of trailing zeros in the binary representation of the determinant. To see whether an edge (u_i, v_j) is in the minimum-weight perfect matching, we set the weight of this edge to 0. If the new minimum weight is $w^* - w_{ij}$, then our edge is in this matching. Each of these tests can be done in parallel.

Perfect Matching is in RNC (Contd.)

Algorithm

We define a matrix $A^{G,w}$ whose i,jth elements is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. Recall that $\det A^{G,w} = \sum_{\pi} \sigma(\pi) \Pi_{i=1}^{n} A_{i,\pi(i)}^{G,w}$, which is actually $2^{w(\pi)}$ since we are not concerned about permutations that are not perfect matchings (those terms are zero). Since the minimum weight is unique, let this be w^*. This means all terms of $\det A^{G,w}$ are multiples of 2^{w^*}, and all of them but one will be even multiples of 2^{w^*}. Therefore, 2^{w^*} is the highest power of two that divides $\det A^{G,w}$, which means we can calculate w^* efficiently in parallel.
We first get $\det A^{G,w}$ using our **NC** algorithm for determinants. w^* will be the number of trailing zeros in the binary representation of the determinant. To see whether an edge (u_i, v_j) is in the minimum-weight perfect matching, we set the weight of this edge to 0. If the new minimum weight is $w^* - w_{ij}$, then our edge is in this matching. Each of these tests can be done in parallel.

Perfect Matching is in RNC (Contd.)

Algorithm

We define a matrix $A^{G,w}$ whose i, jth elements is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. Recall that $\det A^{G,w} = \sum_{\pi} \sigma(\pi)\Pi_{i=1}^{n} A_{i,\pi(i)}^{G,w}$, which is actually $2^{w(\pi)}$ since we are not concerned about permutations that are not perfect matchings (those terms are zero). Since the minimum weight is unique, let this be w^*. This means all terms of $\det A^{G,w}$ are multiples of 2^{w^*}, and all of them but one will be even multiples of 2^{w^*}. Therefore, 2^{w^*} is the highest power of two that divides $\det A^{G,w}$, which means we can calculate w^* efficiently in parallel.
We first get $\det A^{G,w}$ using our **NC** algorithm for determinants. w^* will be the number of trailing zeros in the binary representation of the determinant. To see whether an edge (u_i, v_j) is in the minimum-weight perfect matching, we set the weight of this edge to 0. If the new minimum weight is $w^* - w_{ij}$, then our edge is in this matching. Each of these tests can be done in parallel.

Perfect Matching is in RNC (Contd.)

Algorithm

We define a matrix $A^{G,w}$ whose i, jth elements is $2^{w_{ij}}$ if (u_i, v_j) is an edge, and 0 otherwise. Recall that $\det A^{G,w} = \sum_{\pi} \sigma(\pi) \Pi_{i=1}^{n} A^{G,w}_{i,\pi(i)}$, which is actually $2^{w(\pi)}$ since we are not concerned about permutations that are not perfect matchings (those terms are zero). Since the minimum weight is unique, let this be w^*. This means all terms of $\det A^{G,w}$ are multiples of 2^{w^*}, and all of them but one will be even multiples of 2^{w^*}. Therefore, 2^{w^*} is the highest power of two that divides $\det A^{G,w}$, which means we can calculate w^* efficiently in parallel.
We first get $\det A^{G,w}$ using our **NC** algorithm for determinants. w^* will be the number of trailing zeros in the binary representation of the determinant. To see whether an edge (u_i, v_j) is in the minimum-weight perfect matching, we set the weight of this edge to 0. If the new minimum weight is $w^* - w_{ij}$, then our edge is in this matching. Each of these tests can be done in parallel.

Perfect Matching is in RNC (Contd.)

Random Component

If we randomly assign small weights to the edges, we get a high probability that the minimum-weight matching is unique.

Lemma

Suppose that the edges in E are assigned independently and randomly weights between 1 and $2|E|$. If a perfect matching exists, then with probability at least $\frac{1}{2}$ the minimum-weight perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Random Component

If we randomly assign small weights to the edges, we get a high probability that the minimum-weight matching is unique.

Lemma

Suppose that the edges in E are assigned independently and randomly weights between 1 and $2|E|$. If a perfect matching exists, then with probability at least $\frac{1}{2}$ the minimum-weight perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Random Component

If we randomly assign small weights to the edges, we get a high probability that the minimum-weight matching is unique.

Lemma

Suppose that the edges in E are assigned independently and randomly weights between 1 and $2|E|$. If a perfect matching exists, then with probability at least $\frac{1}{2}$ the minimum-weight perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$

We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge *e*. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain *e*, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain *e*, but not including the weight of *e*. Let $\Delta = w^*[\bar{e}] - w^*[e]$

We now get the weight w_{ij} of *e*. *e* is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then *e* is in every minimum-weight matching, and if $w_{ij} > \Delta$, then *e* is not in any minimum-weight matching. If $w_{ij} = \Delta$, then *e* is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$

We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$

We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$
We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$
We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$
We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$
We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$
We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$
We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**$[e \text{ is bad}] \leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$
We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. $\square$

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Perfect Matching is in RNC (Contd.)

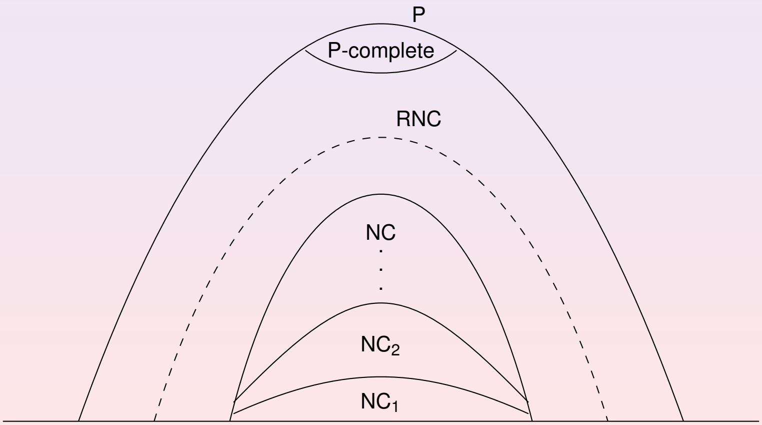
Proof.

We call an edge bad if it is in one minimum-weight matching but not in the other, which means the minimum-weight perfect matching is unique if and only if there are no bad edges. Suppose all weights have been assigned except for edge e. Let $w^*[\bar{e}]$ be the smallest weight among all perfect matchings that do not contain e, and let $w^*[e]$ be the smallest weight among all perfect matching that do contain e, but not including the weight of e. Let $\Delta = w^*[\bar{e}] - w^*[e]$
We now get the weight w_{ij} of e. e is bad if and only if $w_{ij} = \Delta$. This is because if $w_{ij} < \Delta$, then e is in every minimum-weight matching, and if $w_{ij} > \Delta$, then e is not in any minimum-weight matching. If $w_{ij} = \Delta$, then e is bad since both matchings are now minima. It follows that **prob**[e is bad] $\leq \frac{1}{2|E|}$ because this is the probability that a randomly drawn integer between 1 and $2|E|$ will coincide with Δ. This means the probability that there is some bad edge among the $|E|$ ones is at most $|E|$ times that bound, which is no more than half. □

Summary

Assign random weights to the edges, and run the algorithm that computes the minimum-weight matching if it is unique. We have at least a probability of $\frac{1}{2}$ that our perfect matching is unique.

Diagram


P
P-complete
RNC
NC
.
.
.
NC$_2$
NC$_1$