April 1984

Report No. STAN-G-84-1003

# Parallelism and Greedy Algorithms

by

Richard Anderson and Ernst Mayr

Department of Computer Science

Stanford University
Stanford, CA 94305

# Parallelism and Greedy Algorithms

Richard Anderson*

Ernst Mayr

Computer Science Department
St anford University
Stanford, California 94305

Abstract

A number of greedy algorithms are examined and are shown to be probably inherently sequential. Greedy algorithms are presented for finding a maximal path, for finding a maximal set of disjoint paths in a layered dag, and for finding the largest induced subgraph of a graph that has all vertices of degree at least $k$. It is shown that for all of these algorithms, the problem of determining if a given node is in the solution set of the algorithm is F-complete. This means that it is unlikely that. these sequential algorithms can be sped up significantly using parallelism.

This research was suppor ted in part by an IBM Faculty Development Award

* Also supported in part by a National Science Foundation Graduate Fellowship

## Introduction

There are two basic approaches to take in designing parallel algorithms. One can either look for a new algorithm for. the problem, or one can attempt to speed up a known sequential algorithm. Some sequential algorithms can be sped up substantially with minor modifications that take advantage of parallelism. An example of such a case is matrix multiplication where the standard algorithm can be sped up by having each entry in the product be computed by a different processor. On the other hand, there are many sequential algorithms which appear to be very sequential in nature is the class of greedy algorithms. These algorithms are some of the simplest sequential algorithms. In a certain well defined sense some of these algorithms will be shown to be probably inherently sequential. It is interesting that in some cases, different approaches to the same problem will yield fast parallel algorithms.

## Preliminaries

The standard model of synchronous parallel computation is the P-RAM model[FW]. A P-RAM is a collection $P_1, \ldots, P_n$ of identical processors and a collection $M_1, \ldots, M_m$ of global memory cells. Each processor has its own local registers and is capable of performing the standard arithmetic operations (+, −, ∨, ∧, ¬, but not ×). A processor can also. communicate with any memory cell by reading or writing a value. There is a single program that all processors execute one step at a time. Each processor has a register which contains its processor number and instructions may depend on this number, so different processors may do different things. on the same instruction. For example, if $p_{id}$ is the processor number, then the instruction "if $p_{id}$ mod 2 = 0 then $M_{p_{id}} \leftarrow 1$" sets the even numbered memory locations to 1. The time taken for an algorithm is the number of instructions that are executed. If the unit cost model is used to measure space, the space is the sum of the number of processors and the number of memory cells. There are variants of the P-RAM model which handle conflicting reads and writes to the same global memory cell differently. These variants can be simulated by one another with at most a $\log^r n$ slow down for some small $r$. One interesting class of problems to look at are the problems that can be solved in $O(\log^k n)$ time and $O(n^j)$ space on a P-RAM. These problems exhibit an exponential degree of speed up while using a reasonable number of processors. This class is Ireferred to as NC [P].

The parallel computation thesis [FW] is that parallel time is roughly the same as sequential space. $\mathrm{PTIME}(T)$ and $\mathrm{DSPACE}(S)$ denote the class of problems that can be solved in parallel time $T$ and in deterministic sequential space S respectively. (The model of sequential computations is that of multitape Turing machines). The relations between these classes are $\mathrm{PTIME}(T) \subseteq \mathrm{DSPACE}(T^2)$ and $\mathrm{DSPACE}(S) \subseteq \mathrm{PTIME}(S)$ for $S \geq \log n$ [FW) . These results are proved by simulating a parallel computation by a space efficient sequential computation and a sequential computation by a fast parallel computation. A related result is that DLOCSPACE $\subseteq \mathcal{NC}$, where DLOGSPACE denotes the problems that can be solved on a multitape Turing machine in logarithmic space. This result doesn't follow immediately from $\mathrm{DSPACE}(S) \subseteq \mathrm{PTIME}(S)$, since it must also be

shown that the number of processors required for the simulation is polynomial. The result for the other direction is that $\mathcal{NC} \subseteq$ DPOLYLOGSPACE where DPOLYLOGSPACE is $\cup_{k\geq 0}$DSPACE(log" $n$) problems that can be solved in log" $n$ space. The space efficient simulation of $\mathcal{NC}$ problems generally does not give polynomial time algorithms. However, $\mathcal{NC} \subseteq \mathcal{P}$, since a polynomial number of processors can be simulated by a single processor with a polynomial slowdown, using possibly polynomial space.

The parallel computation thesis implies that the most difficult problems to parallelize are the same as the problems that are the most difficult to compute in a small amount of space. Complexity theory provides the notion of completeness which allows the most difficult problems in a class to be identified [GJ], [HU]. A problem is *complete* for a class if it is in the class and all problems in the class can be reduced in an appropriate manner to it. This means that a solution to the complete problem provides solutions to all problems in the class. The most difficult problems in $\mathcal{P}$ to compute in a small amount of space are the problems that are logspace complete for $\mathcal{P}$, or P-complete for short. For defining P-completeness, logspace reductions *are* used. A *logspace reduction* of problem *A* to problem $B$ is a transformation which converts an instance of A into an equivalent instance of $B$ and uses only logarithmic space for the computation. Logarithmic space only counts the work space that the computation uses, read only input and write only output are provided and do not count in the space bound. We will use $A \propto_{\log} B$ to denote that there exists a logspace reduction of *A* to *B*. Logspace transformations are transitive, so if $A \propto_{\log} B$ and $B \propto_{\log}$ C, then $A \propto_{\log} C$. This implies that if a P-complete problem is solvable $O(\log^k n)$ space, $k \geq 1$, then all problems in $\mathcal{P}$ are solvable in $O(\log^k n)$ space. Also, once a P-complete problem is known, other problems in $\mathcal{P}$ can be shown to be P-complete by reducing the P-complete problem to-them.

If a problem is shown to be P-complete, then it is unlikely that there is a fast parallel algorithm for it. Logspace transformations can be done in log $n$ time on a P-RAM using a polynomial number of processors so the P-complete problems are the most difficult problems to parallelize. If a P-complete problem is found to be in $\mathcal{NC}$, then $\mathcal{P} = \mathcal{NC}$ and $\mathcal{P} \subseteq$ DPOLYLOGSPACE. If this were the case, then all problems in $\mathcal{P}$ could be solved very fast in parallel and could be solved sequentially using very little space. Both of these are considered to be very unlikely. There is of course,. as of this writing, no known proof that $P \neq \mathcal{NC}$ as there is no known proof that $P \neq \mathcal{NP}$.

Quite a few problems are known to be P-complete. The central such problem seems to be the circuit value problem [L]. It is:

- given a boolean circuit and values for its inputs, compute the value of its output.

The generic reduction to show that the circuit value problem is P-complete is given a polynomial time Turing machine *M*, and input x, a circuit is constructed in logspace that when given input x will have output 1 if and only if *M* accepts input x. The circuit constructed simulates the Turing machine configurations at each time step. The circuit value problem plays a similar role for P-completeness as satisfiability does for NP-completeness. Most P-completeness proofs are reductions from variants of the circuit value problem. Three important problems that are known to be P-complete are computing the value of the maximum flow in a network [GSS], linear programming [DLR], and unification [DICM].

One of the drawbacks of P-completeness is that it is defined strictly in terms of

language recognition problems. There are quite a few problems which can not be phrased as language recognition problems in any reasonable manner. An example of such a problem is the problem of finding a maximal independent set in a graph. This problem is given an undirected graph, find a set $I$ of vertices which contains no two adjacent vertices and every vertex of the graph not in $I$ is adjacent to some member of $I$. This problem must be distinguished from the maximum independent set problem which is to find a largest cardinality independent set. It would be difficult to express the maximal independent set problem in terms of language recognition. For example the problem of the existence of a maximal independent set is trivial while the existence of such a set with a certain property might be a much more difficult problem.

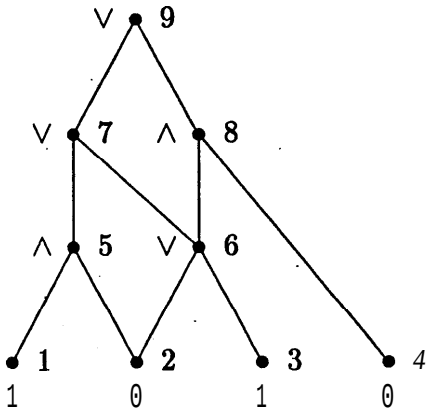
The maximal independent set problem is of the general form that we have a relation $R(x,y)$ and want to find a function $f$ such that $R(x,$ f(x)) for all x in the domain. One approach to showing such a problem to be difficult would be to find a logspace Turing reduction from a P-complete problem to it. A Turing reduction of problem $A$ to problem $B$ is a Turing machine that solves problem $A$ assuming that there exists a Turint machine that solves problem $B$ that it can call as a subroutine. A second approach which is weaker is to show that a specific function or algorithm that solves the problem is "P-complete". With a function $f$, there is an associated language $L_f$ = {(x, a, $i$)| the i-th symbol of the output $f$(x)is a}. A function $f$ will be said to be P-complete if the corresponding language $L_f$ if P-complete. Results of this form show that a particular approach to a problem is not likely to yield an efficient parallel algorithm. Showing that a specific sequential algorithm probably can't be sped up does not imply that the problem itself is difficult. The greedy algorithm for computing a maximal independent set is to consider the nodes in a specific order and add them to an independent set if possible and discard them otherwise. Cook has shown that the language associated with this problem is P-complete [C]. Wigderson and Karp have found a polylog algorithm which outputs a maximal independent set[KW]. Another problem which has a fast parallel algorithm even though its natural greedy algorithm appears to be inherently sequential will be discussed in this paper.

## Greedy Algorithms

Greedy algorithms are some of the simplest sequential algorithms.. A greedy algorithm solves a problem by building its solution set one element at a time. Once an element is added to the solution, it will not be removed, so that there is no backtracking. The rule that adds elements to the solution set is generally fairly simple. Greedy algorithms appear to be very sequential since the decision whether to add an element to the solution set depends on all previous decisions. The following results will show that the problem of computing the same result as the greedy algorithm gives, is P-complete for a number of simple graph problems. This is evidence that some greedy algorithms probably can not be parallelized, and different approaches will have to be used if these problems are to be solved by fast parallel algorithms.

A problem is shown to be P-complete by giving a logspace reduction from a known. P-complete problem to it. All of the reductions in this paper will be from the circuit

value problem. The circuit value problem is given a circuit made up of boolean gates and specified inputs, compute the value of the circuit's output. A circuit $\beta$ is a string $\beta_1, \ldots, \beta_n$, where $\beta_i$ is either an input with value 0 or 1, or a boolean gate from some basis such as {AND, OR, NOT}. The gates are numbered topologically so if $\beta_k$ has inputs $i$, j then $i < k$ and j < $k$. An example of a circuit is shown below with both the numbers and the types of the gates shown. Many variants of the circuit value problem are known to be P-complete [G]. Two P-complete circuit problems that will be used in this paper are the monotone circuit value problem (MCVP) and the planar circuit value problem (PCVP). Circuits can be restricted to be only made up of AND and OR gates giving the monotone circuit value problem. The outdegree of a gate can without loss of generality, be limited to being at most two. It is quite easy to simulate higher fanout using extra circuit elements. For another variant, the circuit can be restricted to being planar, i.e. its underlying graph is planar, and it contains only NOT and OR gates. The circuit can be further restricted so that the gates can be laid out in levels with connections only going to adjacent levels. The problem remains P-complete with these restrictions [G].




Example circuit
The circuit is encoded as $(1,0,1,0,1 \wedge 2, 2 \vee 3, 5 \vee 6, 6 \wedge 4, 7 \vee 8)$

## The Lexicographic Path Problem

The dead end path problem is:

given a graph G = *(V, E)* and a distinguished node $r$, find a simple path starting from $r$ that cannot be extended without going to a vertex that is already on the path.

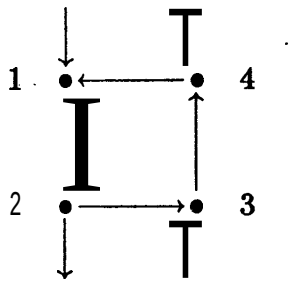
The greedy algorithm for finding a dead end path is to start at the root and always go to the lowest numbered unvisited neighbor. When the algorithm stops a dead end path has been found. If the edges are represented by adjacency lists, then the greedy algorithm will go to the first element in the list that has not been visited. If the adjacency lists are sorted then the algorithms are the same. The algorithm is:

**Algorithm** Greedy

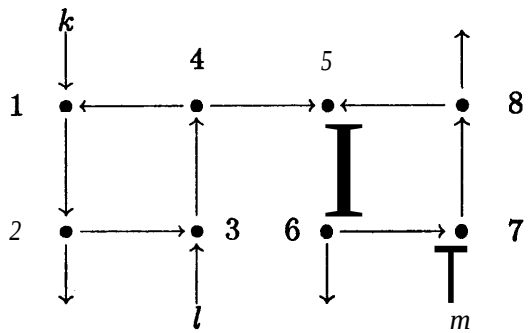
```
path ← v0; v ← v0;
while v has an unvisited neighbor do
    w ← lowest numbered unvisited neighbor of v;
    path ← path followed by w;
    v ← w
end
```

The natural lexicographic ordering on paths is p $<_{lex}$ $q$ if p and $q$ are the same for the first $k-1$ positions and $p_k = \epsilon$ or $p_k < q_k$. The path formed by the greedy algorithm is the lexicographically minimum dead end path. The lexicographic path problem (LPP) is to determine whether the vertex $v_n$ is on the lexicographically minimum dead end path. The problem LPP will be shown to be P-complete. This result applies to directed graphs (DLPP), undirected graphs (ULPP), and planar graphs (PLPP). We will first show that DLPP is P-complete and then give the modifications to the construction to show that the undirected and planar cases are also P-complete. This will show that there is little hope of finding a fast parallel algorithm that computes the same dead end path as the sequential algorithm.

To show that DLPP is P-complete we will give a logspace reduction from MCVP to DLPP. To do this we must show how we can simulate a circuit with a path in a. graph. Let $\beta$ be a monotone circuit. Each gate of $\beta_k$ will be simulated by a collection of nodes. The gates will be simulated in order, with the path testing inputs and setting outputs as appropriate. The gate $\beta_k$ will be simulated by a path which will go from node $k_{in}$ to $k_{out}$. The key component of the simulation is a switch which is used to simulate the value of a wire. If a switch is traversed from the corresponding gate, then it will indicate a true value. When a gate tests a value of a switch it will traverse the switch if it was not set, and will not be able to enter it if it was set. A switch is:




If the switch is entered from node 1, it will be left from node 4, and if it is entered from 3, it will be left from 2. The nodes adjacent to nodes 2 or 4 that are outside the switch have labels greater than 4. This prevents the path from leaving the switch before it has visited all the nodes of the switch. Two switches may be needed for the outputs of a gate, so they are joined as in the figure below. Note that the numbering insures that both switches are visited if entry is made from $k$, but only 1 switch will be visited if entering from $l$ or $m$.



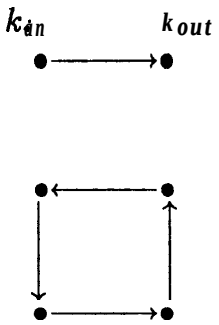

**Theorem :**

The directed lexicographic path problem is P-complete.

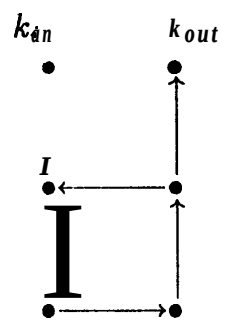
**Proof:** The proof is a logspace reduction of MCVP to DLPP. Let $\beta = \{\beta_0, \beta_1, \ldots, \beta_n\}$ be a monotone circuit. A graph G will be constructed such that the node $v_n$ is visited by the lexicographic path starting from $v_0$ iff the value of the gate $\beta_n$ is 1.

Each gate $\beta_k$ corresponds to a subgraph with nodes $k_{in}$ and $k_{out}$. If $k \neq n$, $k_{out}$ will be connected to $(k + 1)_{in}$. The four types of gates are O-INPUTS, l-INPUTS, AND gates, and OR gates. The subgraphs for these gates are illustrated in the figures below. If $\beta_k$ is an AND or an OR gate with inputs from $i_x$ and $j_y$, x, y $\in \{1, 2\}$, it is connected to switches x and y of gates $i$ and j. The nodes are numbered so that all nodes associated with gate $k$ are greater than the nodes associated with gate j for $k >$ j. In the illustrations, the nodes labeled $k$, $k_1$, and $k_2$ are numbered so that $k < k_1 < k_2$.

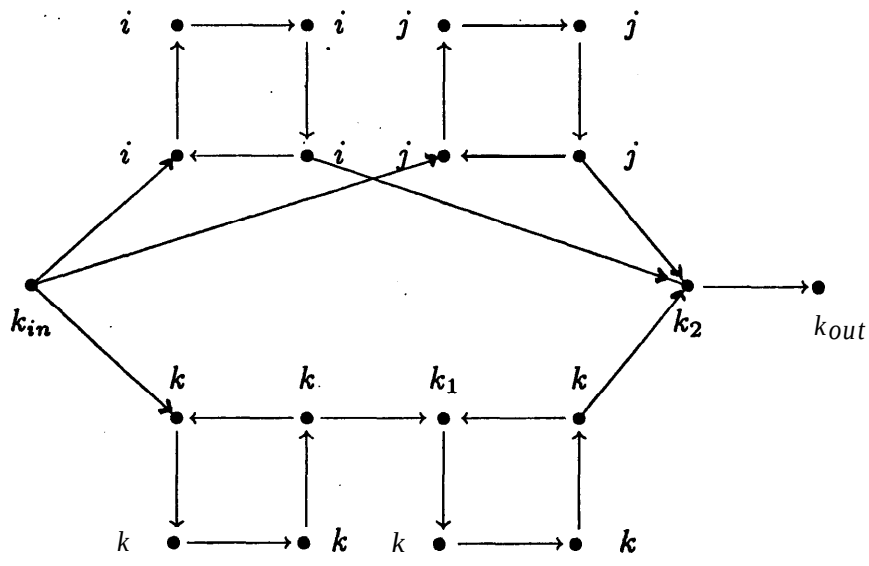
The lexicographically minimum path from $0_{in}$ will set the switches for the inputs that are one and will not set the switches for inputs that are zero. It is straight forward to verify that the path will go through $k$'s switch(es) if $k$ is an AND gate and both of its input switches were set by earlier gates, or if $k$ is an OR gate and at least one of its inputs was set by an earlier gate, and the switch(es) will not be visited otherwise. Hence, the switch for gate $n$ will be visited iff the value of $\beta_n$ is 1. ∎



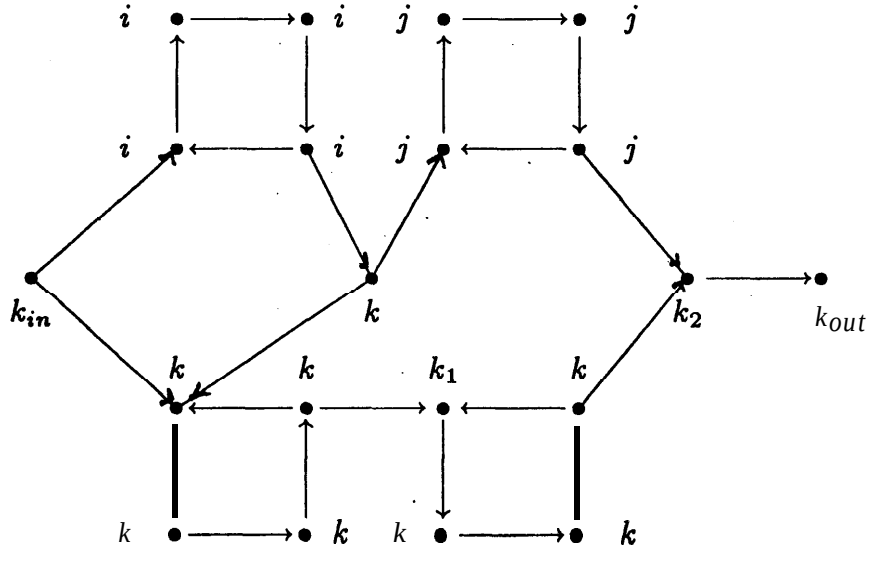

O-INPUT




1 - INPUT




AND gate



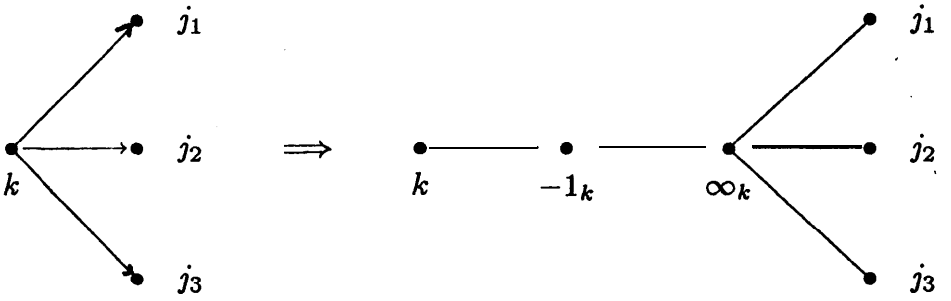

OR gate

**Theorem :**

The undirected lexicographic path problem is P-complete.

**Proof:** The same reduction is used as in the directed case except with a minor modification so that undirected edges will simulate directed edges. The node $k$ is replaced by three nodes numbered $k$, $-1_k$, and $\infty_k$. There is an edge between $k$ and $-1_k$ and between $-1_k$ and

$\infty_k$. Every directed edge from $k$ to some j is replaced by an edge from $\infty_k$ to $j$. When a node $k$ is visited, the next node visited will be $-1_k$ and then $\infty_k$ and then the node that would have been visited in the directed case. The nodes numbered $\infty$ prevent edges from being traversed in the opposite direction from the directed case. The modification is illustrated below. ∎




Converting directed edges to undirected edges

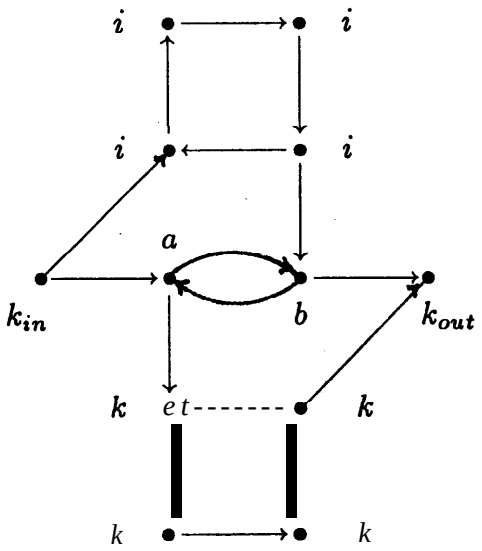
**Theorem :**

The planar lexicographic path problem is P-complete.

**Proof:** This proof is a reduction from PCVP to PLPP. The reduction is very similar to the previous ones. The circuit is assumed to be planar and laid out in levels. The connections only go between consecutive levels. The gates are inputs, NOT gates, and OR gates. The path simulating the gates traverses one level at a time, reversing direction after each level. Since the connections are only between adjacent levels, the edges between gates will not cross the connections. The structures simulating the inputs and the OR gate are the same as for the directed case. The NOT gate is shown below. The NOT gate avoids having to have an edge crossing by using the anti-parallel edges between the nodes a and b. The nodes a and $b$ are numbered less than $k$ and $k_{out}$ and greater than $i$. If node a is entered before $b$, then the path will go to $b$ and then to $k_{out}$, if $b$ is entered first, then the path will go to a and then to the switch. This circuit can be converted to an undirected circuit by a similar method as was used above. There is an exception that the two nodes a and $b$ must be handled differently. The pair of anti-parallel edges between them and the edges out of a and $b$ can just be replaced by undirected edges. ∎

**Finding a dead end path in a planar graph**

Although the greedy algorithm for finding a dead end path in a planar graph is P-complete, a different approach yields an efficient parallel algorithm for finding a dead end path in a planar graph. The basic method is to find a path which divides that graph into



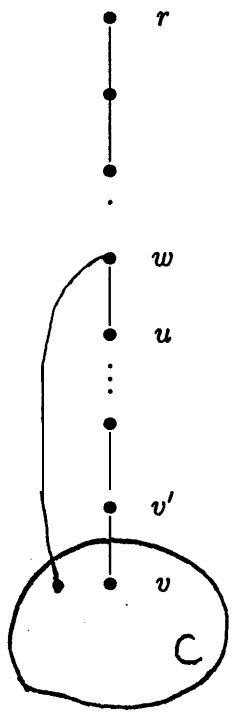

Planar NOT gate

two components. This path can be used to reduce the problem to a problem of half the size of the original problem. The situation is the same as for the maximal independent set problem where the greedy algorithm probably can not be sped up significantly, while a different approach can be used to get a parallel algorithm. It is not known if there is an algorithm which will find a dead end path in a general graph in polylog time:

Let G = (V, E) be a planar graph, and $r \in V$ The key observation for our algorithm is that if a path from $r$ separates the graph into more. than two components, then the problem can be reduced to a problem of less than half the size of the original problem. The path can be followed from $r$ to the last node that is adjacent to two of the components. The path can then be made to go into the smaller of the components. Since the path will never leave the component, we have reduced the problem to solving a much smaller problem.

To split the graph we shall pick a node v and attempt to visit all of its neighbors. Since the graph is planar, we can find a node v with degree at most 5. First we can make sure that the root $r$ has degree at least two. If $r$ is an articulation point, then the trivial path just containing $r$ splits the graph. If the graph has an articulation point different from $r$, we can find an articulation point v that is adjacent to at least two members of the biconnected component containing $r$. We then find the shortest path from $r$ to v which will clearly split the graph into at least two components. If the graph does not have an articulation point, we pick a node v that has lowest degree. Let $v_1, \ldots, v_m$ be the neighbors of v. We temporarily delete the node v from the graph. We then find a path from $r$ to $v_1$, then from $v_1$ to an unvisited neighbor of v, and from that neighbor to another neighbor and so on, never visiting any node more than once. When we can no longer visit any neighbors, we add v to the path. There are three cases that can occur. First, if we have visited all of v's neighbors, we have found a dead end and we are done. Otherwise if the path splits the graph into at least two components; then we can reduce the problem. The. other case is when all the unvisited nodes are in a single component and some neighbor

of v was not visited. This case is illustrated by the figure below. The last node visited before v was v'. C is the connected component of vertices not on the path. The node $v'$ cannot be connected to C or else the path could have been extended. Since v is not an articulation point, there must be au edge from C back to the path. Let $w$ be the last node on the path other than v that has an edge to C and let $u$ be the node that follows w. We can now find a dead end by following the path from $r$ to $w$, then taking the edge into C and then going to v and back up the path to u. All of u's neighbors must be on the path, so we have a dead end.




Finding a dead end path

The individual steps such as finding a path between two vertices, testing for articulation points and finding connected components can all be done in $O(\log^2(n))$ time on $O(n^3)$ processors [V]. Since the problem size is reduced by at least half every time a path is found that splits the graph, no more than logn stages will be required. Since the graph is planar, we will have to find at most 5 paths in each stage. The algorithm therefore runs in $O(log^3(n))$ time on $O(n^3)$ processors.

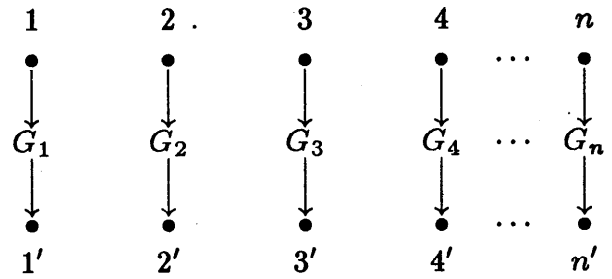
## The maximal set of disjoint paths problem

Another path problem is: find a maximal set of disjoint paths in a layered directed acyclic graph. The motivation for looking at this problem is that it occurs as a subroutine in a number of sequential max flow and matching problems. A layered dag is a dag with

its nodes laid out in levels. The edges only go between consecutive levels. The problem is to find a maximal set of node disjoint paths from the first level to the last level. The greedy algorithm for solving this problem finds paths one at a time and removes them from the graph. This process continues until the first level is separated from the last level. The paths found by the greedy algorithm are the lexicographically minimal paths from the first level to the last level. When a path is removed, it might cause some nodes to be separated from the last level. These nodes are also removed when the path is removed. We will show that the problem of determining if a node lies on one of the paths found by the sequential algorithm is P-complete. It should be noted that the complexity of the problem does not arise from finding the lexicographically minimal path, since that can be done easily in a dag. The complexity arises from the dependence of a path on the previous choices of paths.

In this proof the reduction will be from the circuit value problem. The circuit will be restricted to only contain inputs, NOT gates, and AND gates. The fanout of all gates will be at most 2. For convenience it will be assumed that the fanout is exactly 2, although one of the outputs of a gate might not be connected to anything. The gates are numbered so that inputs to a gate are always from lower numbered gates. The outputs of gate $i$ will be $i_1, i_2$. If $k$ and $k'$ have inputs from $i$ and $k < k'$ then $k$ will get the output $i_2$ and $k'$ will get the output $i_1$. It is also assumed that the AND gates get their inputs from distinct gates, so the situation $k = \mathrm{AND}(i, i)$ is not allowed.

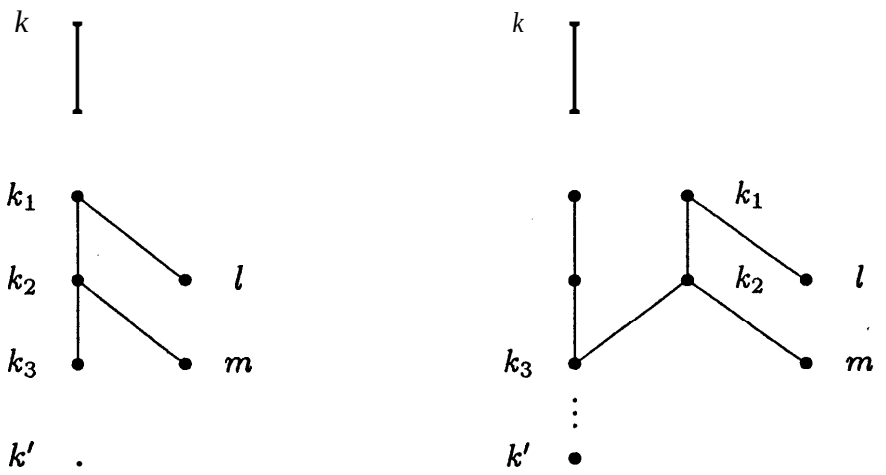
Let $\beta = \beta_1, \ldots, \beta_n$ be a circuit satisfying the above conditions. The basic structure of the dag which simulates the circuit will be:




$G_k$ is a gadget which simulates the gate $\beta_k$. A path $P_k$ will be constructed in $G_k$ which computes the value of the gate, The paths will be constructed in the order $P_1, P_2, \ldots, P_n$. The path $P_k$ will be in $G_k$ except when it tests the inputs of the gate $\beta_k$. In each $G_k$ there will be two special nodes which will be visited by the path $P_k$ if the output of $\beta_k$ is true. These nodes might be visited later if they are not set by $P_k$. The value of the circuit will be true iff the special nodes of $G_n$ are visited.

The graph will have $3n$ levels. The output nodes for $G_k$ will be numbered $k_1$, $k_2$. There will also be a node $k_3$. These nodes will be on levels $3k - 2$, $3k - 1$, and $3k$ respectively. The gadgets for the inputs are illustrated below. In the figures, all edges are directed downwards. Then nodes $l$ and m are on other gadgets. In both cases the path will go directly from $k$ to $k'$. The nodes $k_1$, $k_2$ will be visited if the input is true and passed up otherwise.

The not gate tests its input, and sets its output if the input is false. For the gate $k = \mathrm{NOT}(i_x)$, $x \in \{1, 2\}$, the construct is shown below. The node $l$ is numbered so it




1 - INPUT 0 - INPUT

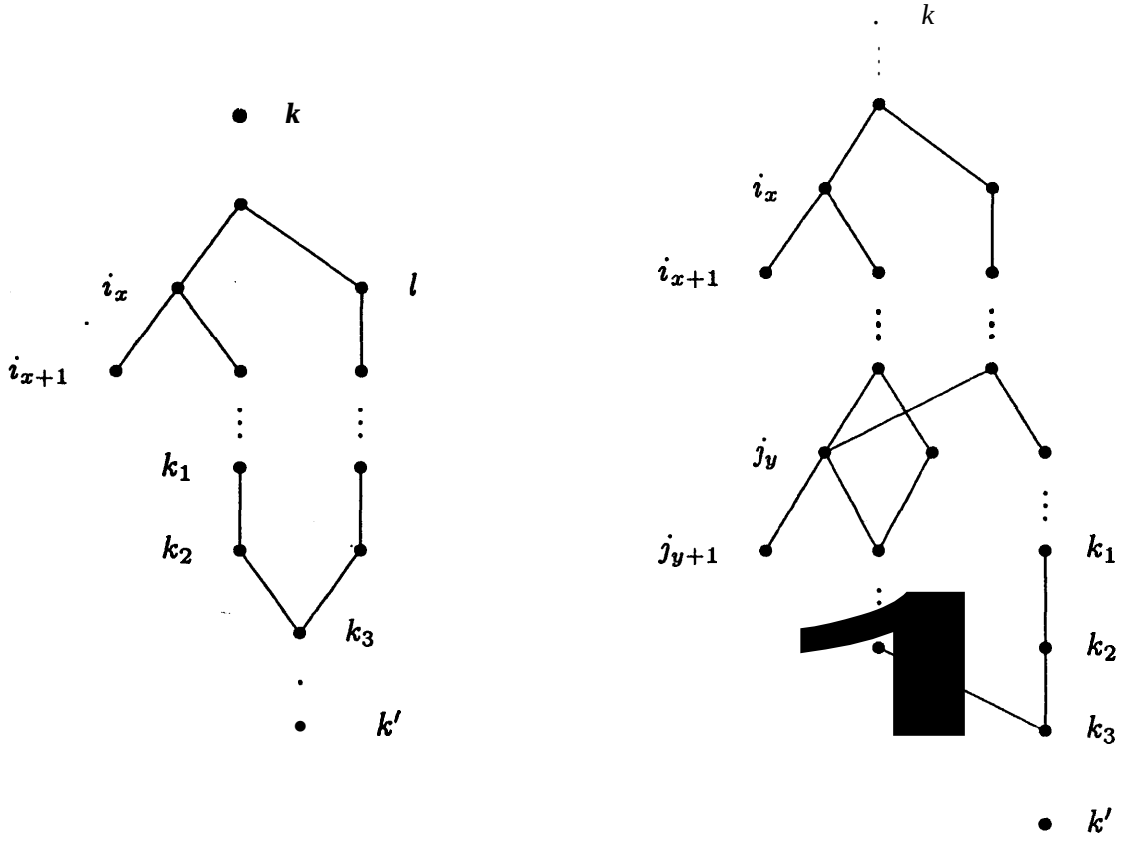
is greater than $i_x$, so if $i_x$ has not been visited, the path will be down through. $k_1$ and $k_2$. The node $i_{x+1}$ will have been visited before $i_x$ is visited so the path will not go out on another gadget. This will be proved below. The AND gate visits both .of the nodes corresponding to its inputs, and if they are both true, then the outputs are set. The gate for $k$ = AND($i_x$, j,) is shown. It is straight forward to verify that $k_1$ and $k_2$ will be set if $i_x$ and $i_y$ are set assuming that $i_{x+1}$ and $j_{y+1}$ have been set previously. Note that even if $i_x$ is false, the path will still attempt to read $j_y$.

For both NOT and AND gates it was critical that if an input is read from $i_x$ that the node $i_{x+1}$ has already been visited. When a gate $k$ is simulated, the node $k_3$ is visited. The first gate that gets an input from $k$ will read $k_2$ since that was one of the conditions put on the circuit. Since the gates read all of their inputs, $k_2$ will be traversed before $k_1$ is read. This completes the proof that this disjoint path problem successfully simulates the circuit.

## . High valency subgraphs

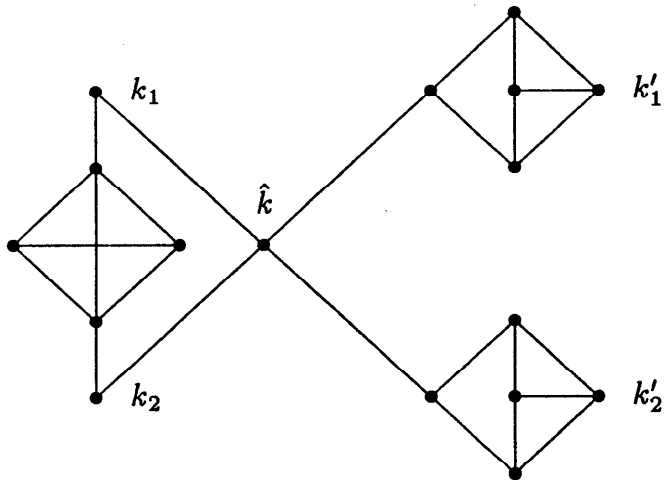
Another problem which is P-complete is finding the maximum induced subgraph of a graph that has all its nodes having degree at least $k$. A lemma by Erdős is that if a graph has $n$ nodes and m edges, then it has an induced subgraph with minimum degree at least $\lfloor \frac{m}{n} \rfloor$ [E]. The proof of this lemma is essentially a constructive algorithm. The algorithm just discards nodes of degree less than $\lfloor \frac{m}{n} \rfloor$ until all nodes have degree at least $\lfloor \frac{m}{n} \rfloor$. It turns out that this set is non-empty. Unfortunately, the problem of determining if a particular node is discarded by this algorithm is P-complete. For a graph G and an integer $k$, the maximum induced subgraph of degree $k$ may be found by discarding nodes until all nodes have degree at least $k$. The maximum induced subgraph could of course be empty.

The proof that the problem of determining if a node $n$ is in the maximum induced

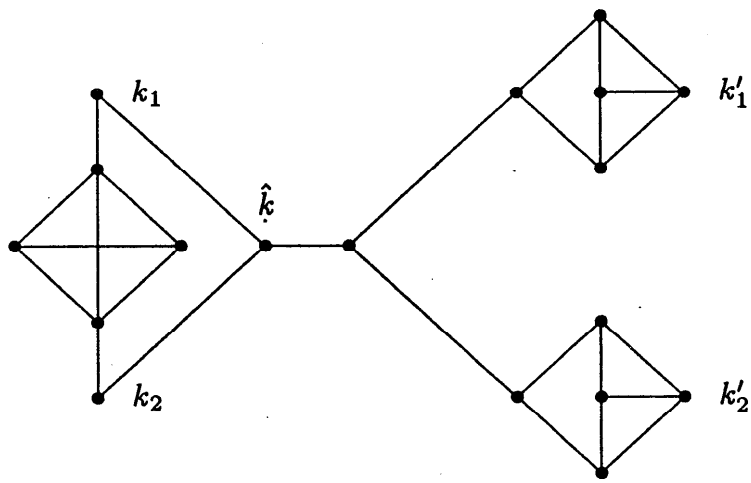



NOT gate AND gate

subgraph of degree $k$ ($k > 2$) is P-complete will be another reduction from a circuit value problem. In this case the reduction will be from the monotone circuit value problem, so the gates are AND gates and OR gates. The proof will be for $k = 3$. A gate will be simulated by a collection of nodes. One of the nodes will give the value of the gate, it will be left in the maximum induced subgraph iff the gate's output is true. The inputs have fanout 1. The false inputs are just an isolated node $k$ which will be removed. A true input consists of 4 nodes, connected to themselves, since they all have degree three, they will never be removed. One of them is distinguished and will be connected to the gate that reads it. The AND gates and OR gates have 4 distinguished nodes $k_1$, $k_2$, $k'_1$, $k'_2$. The nodes $k_1$ and $k_2$ are inputs to the gate and $k'_1$ and $k'_2$ are the outputs of the gates. If the input $k_x$ is connected to the output $j'_y$ of another gate, then there is an edge between $j'_y$ and $k_x$. The gates are shown below. On both of the gates, $k'_1$ and $k'_2$ will only be removed if the node $\hat{k}$ is removed. In the OR gate, $\hat{k}$ will be removed if both $k_1$ and $k_2$ are removed and in the AND gate, $\hat{k}$ is. removed if either $k_1$ or $k_2$ is removed. The nodes $k_1$ and $k_2$ are removed if the connection into them are removed, in other words if they receive a false input. Note that the values will not propagate backwards through the circuit, so a node depends only




OR **gate**




**AND gate**

on the nodes of the gates preceding it. Hence the graph simulates the circuit.

## Conclusions

The major conclusion to be drawn from these results is that greedy algorithms are not likely to yield fast parallel algorithms in some cases. Other approaches will be necessary if efficient parallel algorithms are to be found. It is not known if the dead end path problem or the maximal set of disjoint paths problem can be solved by a fast parallel algorithm.

There are a number of problems where it is not known if the greedy algorithms yield a P-complete problem or not. One particularly interesting problem is finding a maximal matching by the greedy algorithm. This is a special case of finding a maximal independent set with the greedy algorithm since finding a maximal matching is equivalent to finding a maximal independent set in the corresponding line graph. The proof that finding the maximal independent set found by the greedy algorithm is P-complete does not hold for line graphs. It is possible to simulate logical gates with a line graph, the difficulty is that it doesn't appear to be possible to fanout information. The problem of finding a maximal matching with the greedy algorithm exhibits a complex dependency between the various steps of the algorithm, so that it also appears that it would be difficult to find a fast parallel algorithm for this problem.

## References


[C] Cook, S.A., "The Classification of Problems which have Fast Parallel Algorithms", Technical Report No. 164/83, Department of Computer Science, University of Toronto 1983.

[DKM] Dwork, C., Kanellakis, P.C., Mitchell, J.C., "On the Sequential Nature of Unification", 1983.

[DLR] Dobkin, D., Lipton, R.J., Reiss, S., "Linear Programming is Log-Space Hard for P", Information Processing Letters, 8 1979, pp 96-97.

[E] Erdős, P., "On the Structure of Linear Graphs", *Isreal Journal* of *Mathematics*, 1 1963, pp 156-60.

[FW] Fortune, S., Wyllie, J., "Parallelism in Random Access Machines", *Proc. 10th* ACM STOC, 1978, pp 114-118.

[GJ] Garey, M., Johnson, S., Computers and Intractability: A Guide to the Theory of *NP-Completeness*, H. Freeman, San Francisco, 1978.

[G] Goldschlager, L.M., "The Monotone and Planar Circuit Value Problems are Log Space Complete for P", ***SIGACT News*** **9**, 2 1977, pp 25-29.

[GSS] Goldschlager, L.M., Shaw, R.A., Staples, J., "The Maximum Flow Problem is Log Space Complete for P", *Theoretical* Computer Science, **21**, 1982, pp 105-111.

[HU] Hopcroft, J., Ullman, J., *Introduction* to Automata Theory, Languages, and ***Complexity,*** Addison- Wesley, Reading, Mass., 1979.

[KW] Karp, R.M., Wigderson, A., "A Fast Parallel Algorithm for the Maximal Independent Set Problem", 1983.

[L] Ladner, R.E., "The Circuit Value Problem is Log Space Complete for P", SIGACT News 7, 1 1975, pp 583-590.

[P] Pippenger , N., "On Simultaneous Resource Bounds", *Proc.* 20th FOCS, 1979, pp 307-311.

[V] Vishkin, U., "An Optimal Parallel Connectivity Algorithm", RC 9149, IBM T. J. Watson Research Center, Yorktown Heights, N. Y ., 1981.