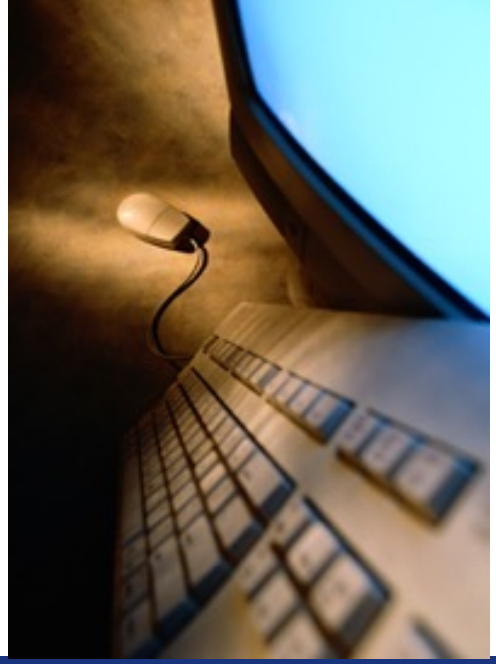
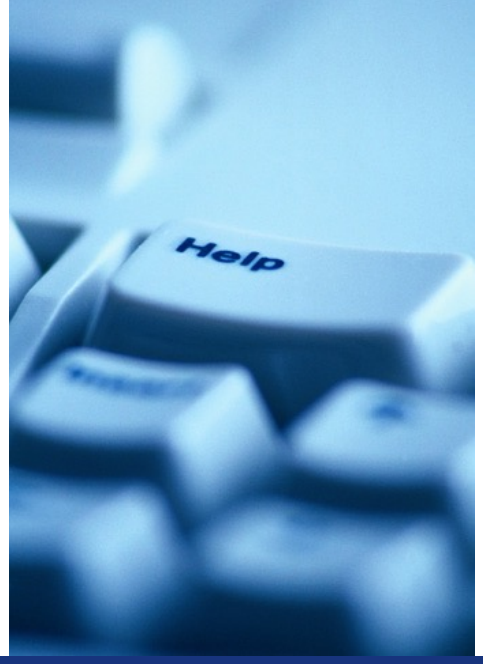
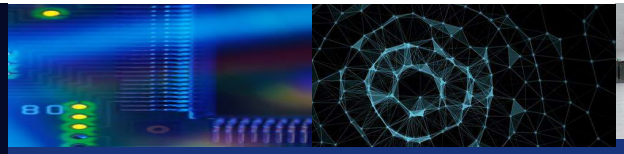



并行与分布式计算

Parallel & Distributed Computing

陈鹏飞
数据科学与计算机学院
chenpf7@mail.sysu.edu.cn



Lecture 13 — Discrete Search & Load Balancing

Pengfei Chen

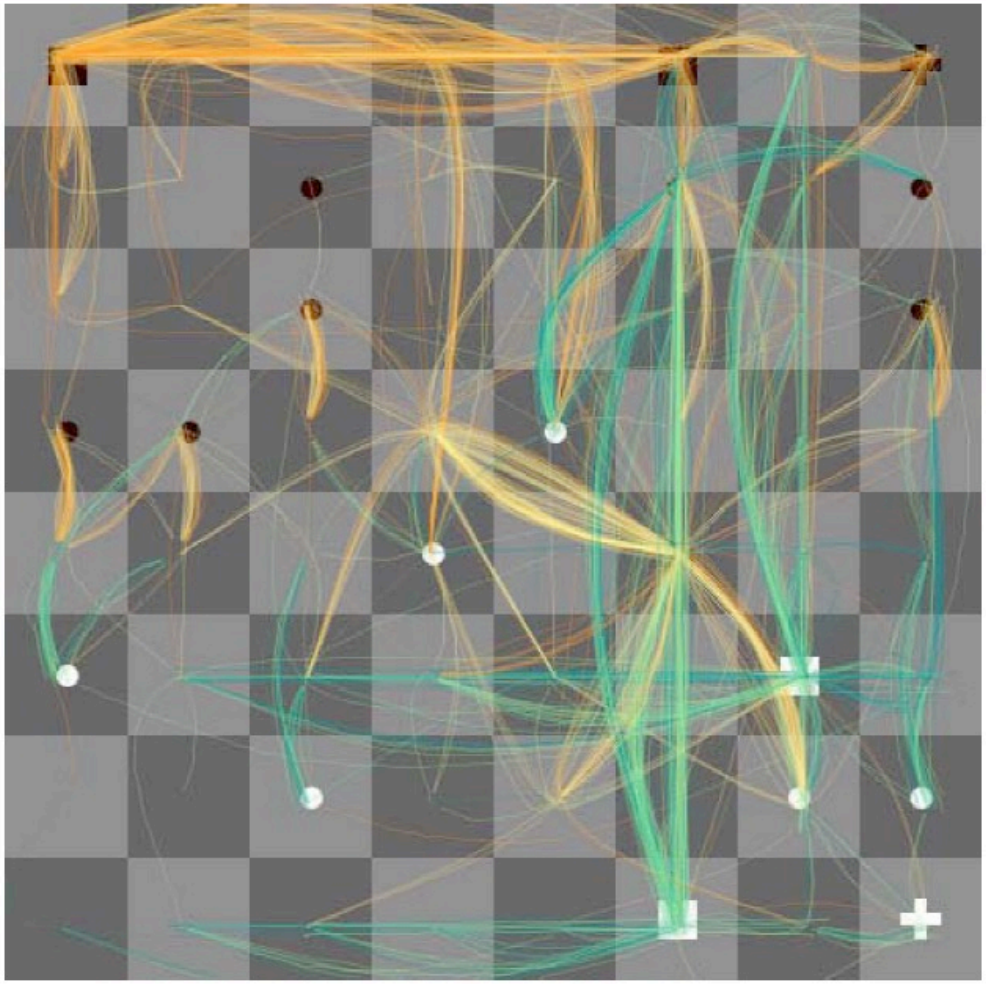
School of Data and Computer Science

chenpf7@mail.sysu.edu.cn

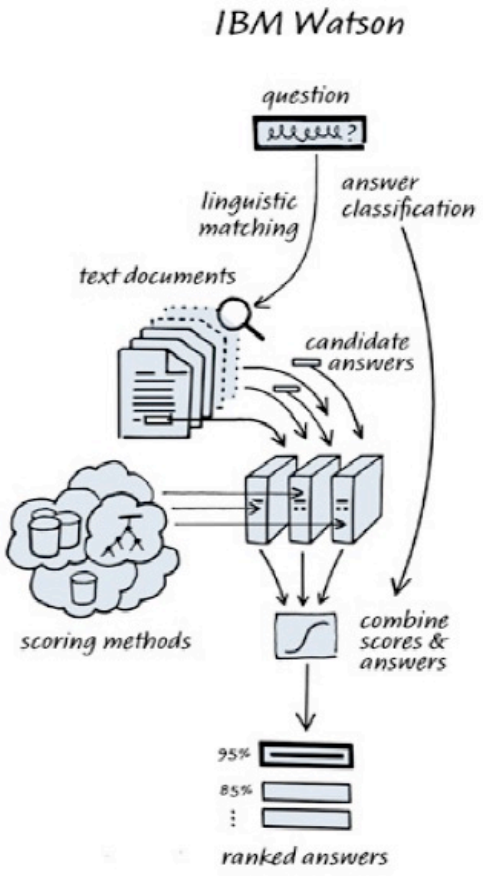
In this Lecture …

- **Parallelization & load balancing schemes**
  - **in depth-first search**
  - **in best-first search**
- **Speedup anomalies**

Example of Discrete Search

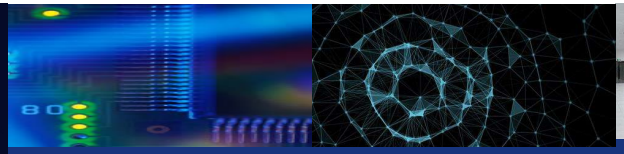


IBM Watson



https://v.qq.com/x/page/k0933twnpx8.html

IBM Watson



Depth-First Search

- Uses depth-first search to consider alternative solutions to a combinatorial search problem
- Recursive algorithm
- Backtrack occurs when
  - A node has no children (“dead end”)
  - All of a node’s children have been explored

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

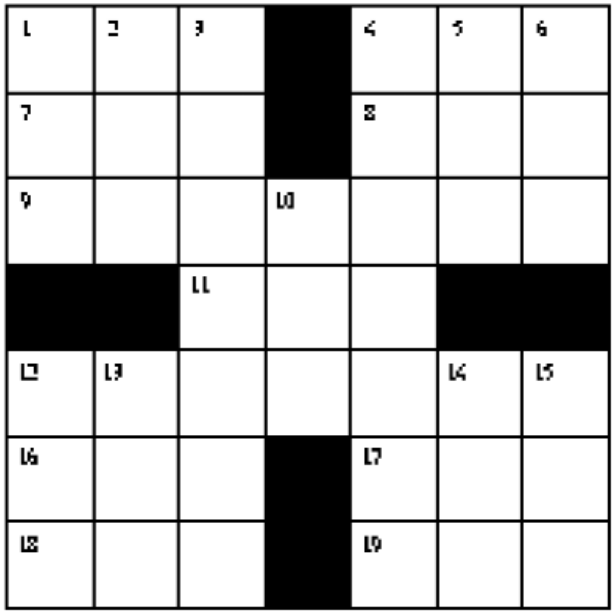
Example: Crossword Puzzle Creation

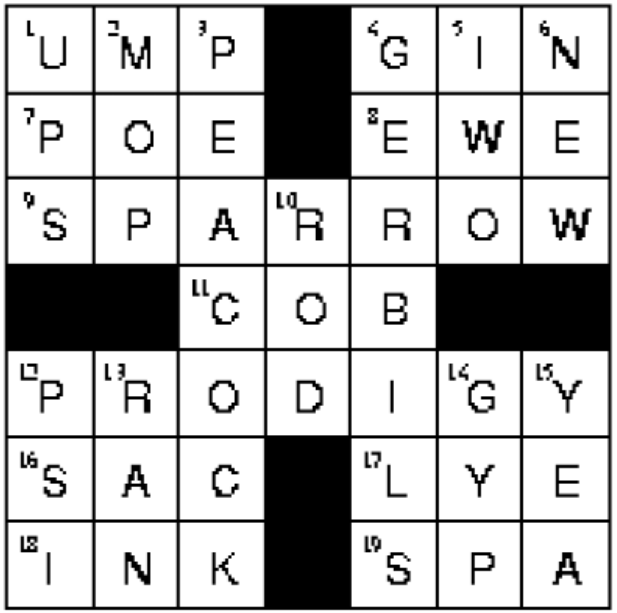
- Given
  - Blank crossword puzzle
  - Dictionary of words and phrases
- Assign letters to blank spaces so that all puzzle's horizontal and vertical "words" are from the dictionary
- Halt as soon as a solution is found

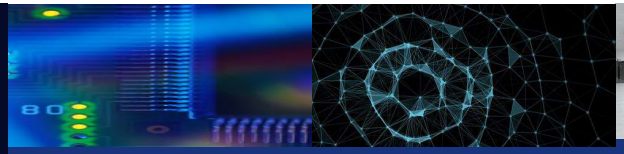
Crossword Puzzle Problem

Given a blank crossword puzzle and a dictionary

find a way to fill in the puzzle.



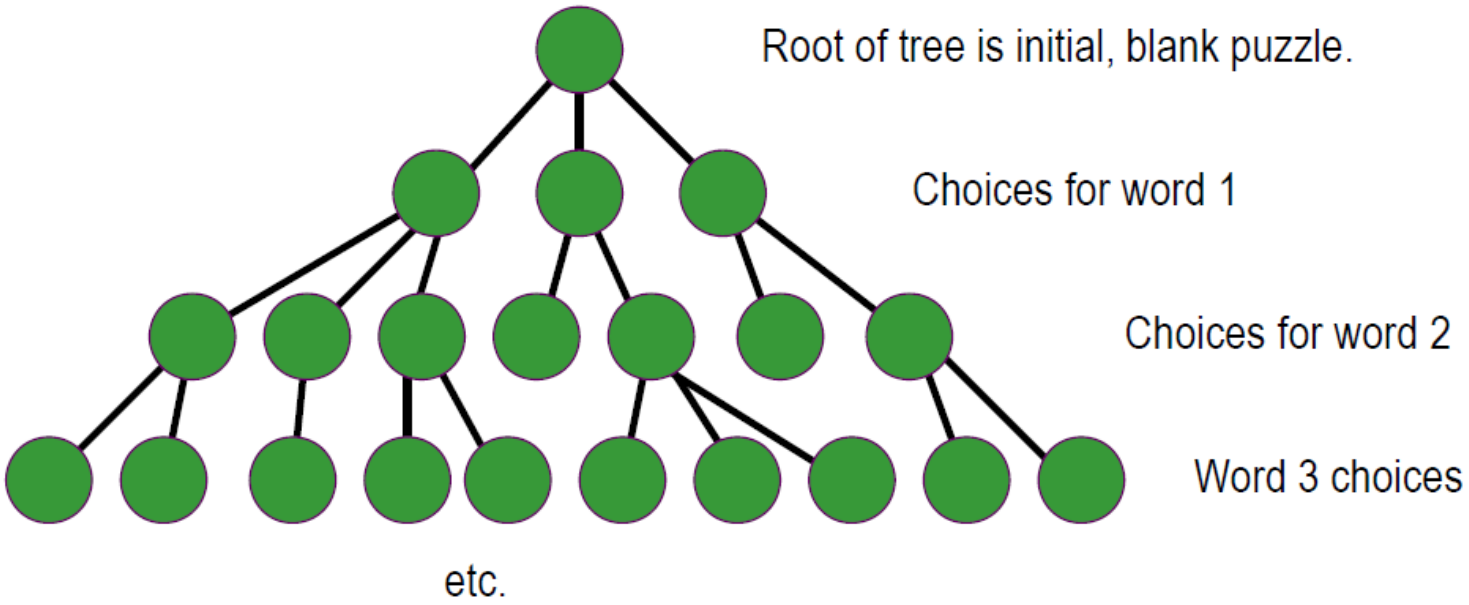




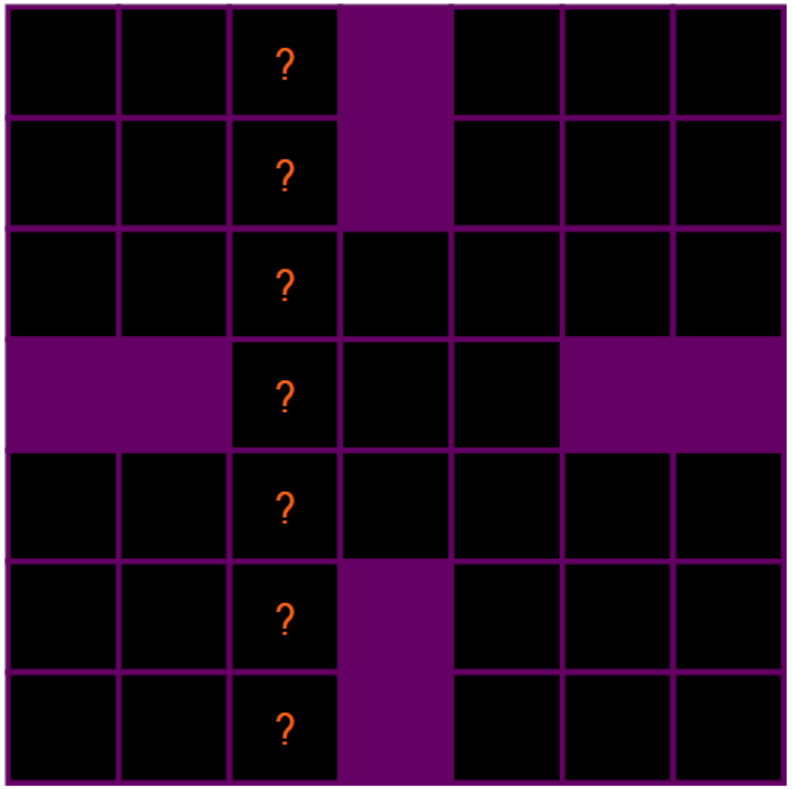
A Search Strategy

- Identify longest incomplete word in puzzle (break ties arbitrarily)
- Look for a word of that length
- If cannot find such a word, backtrack; otherwise,
  - Find longest incomplete word that has at least one letter assigned (break ties arbitrarily)
  - Look for a word of that length
  - If cannot find such a word, backtrack
- Recurs until a solution is found or all possibilities have been attempted

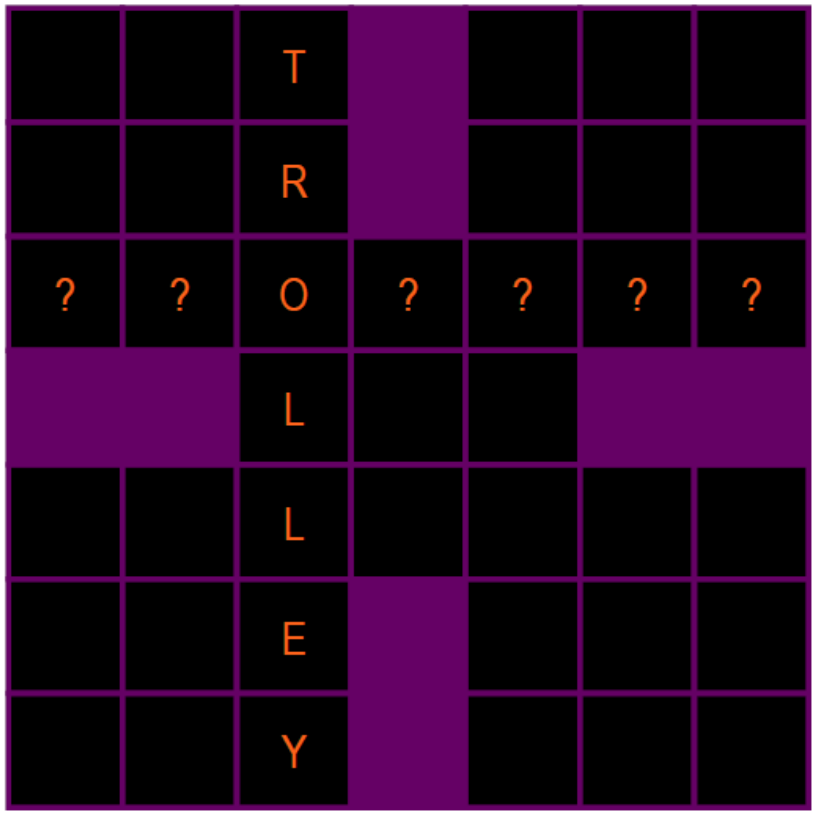
State Space Tree

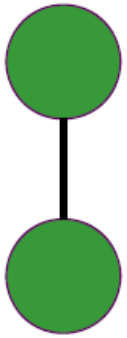


Depth-First Search

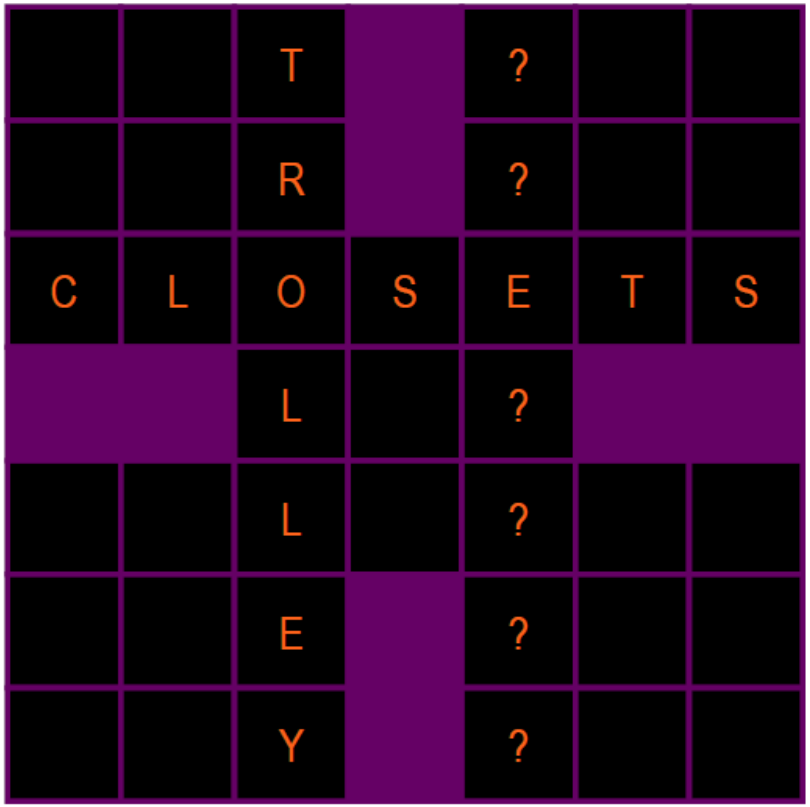


Depth-First Search





Depth-First Search



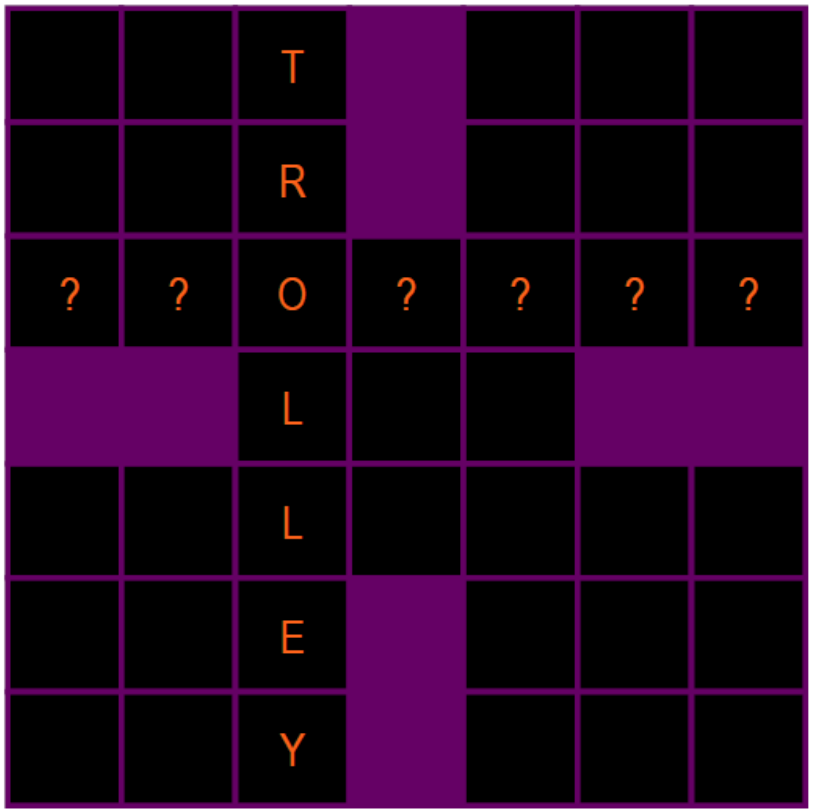
Depth-First Search

		T		?		
		R		?		
C	L	O	S	E	T	S
		L		?		
		L		?		
		E		?		
		Y		?		

Cannot find word.

Must backtrack.

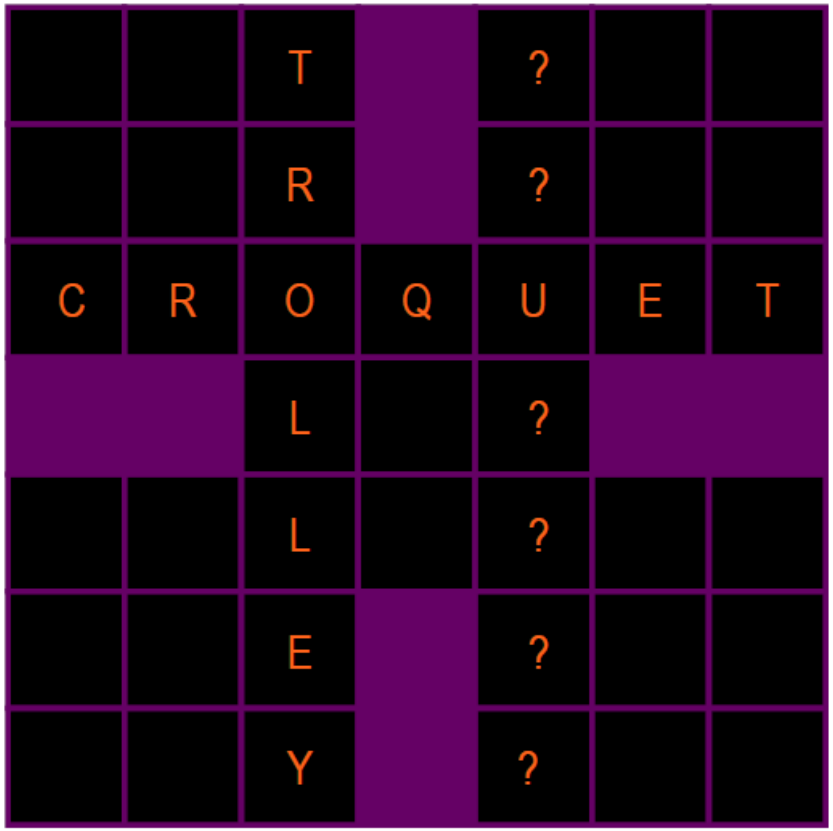
Depth-First Search

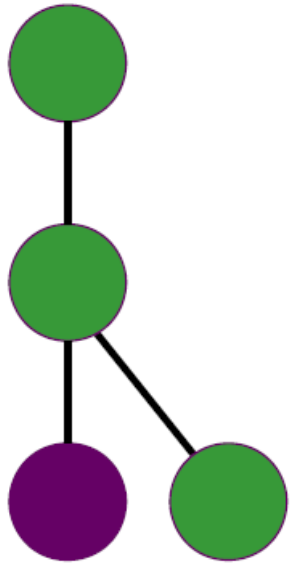


Cannot find word.

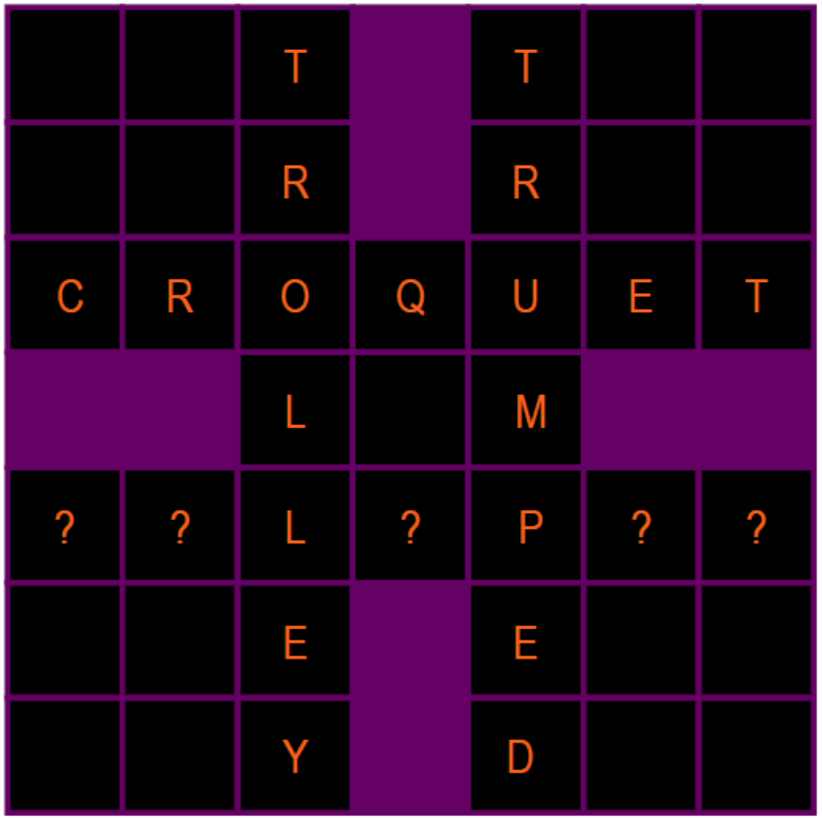
Must backtrack.

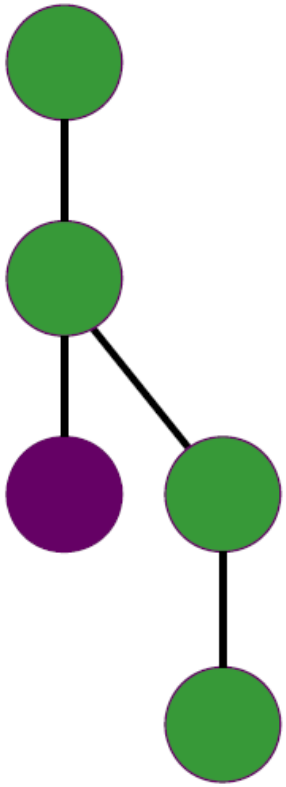
Depth-First Search

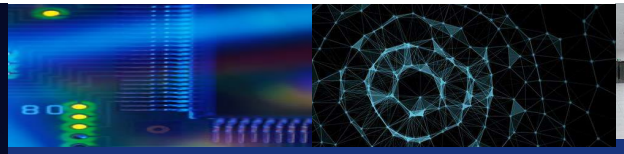




Depth-First Search





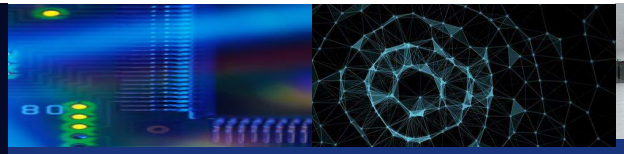


Time and Space Complexity

- Suppose average branching factor in state space tree is b
  - Branching factor: the (average) number of children at each node
- Searching a tree of depth k requires examining

$$1+b+b^2+\cdots+b^k=\frac{b^{k+1}-b}{b-1}+1=\theta(b^k)$$

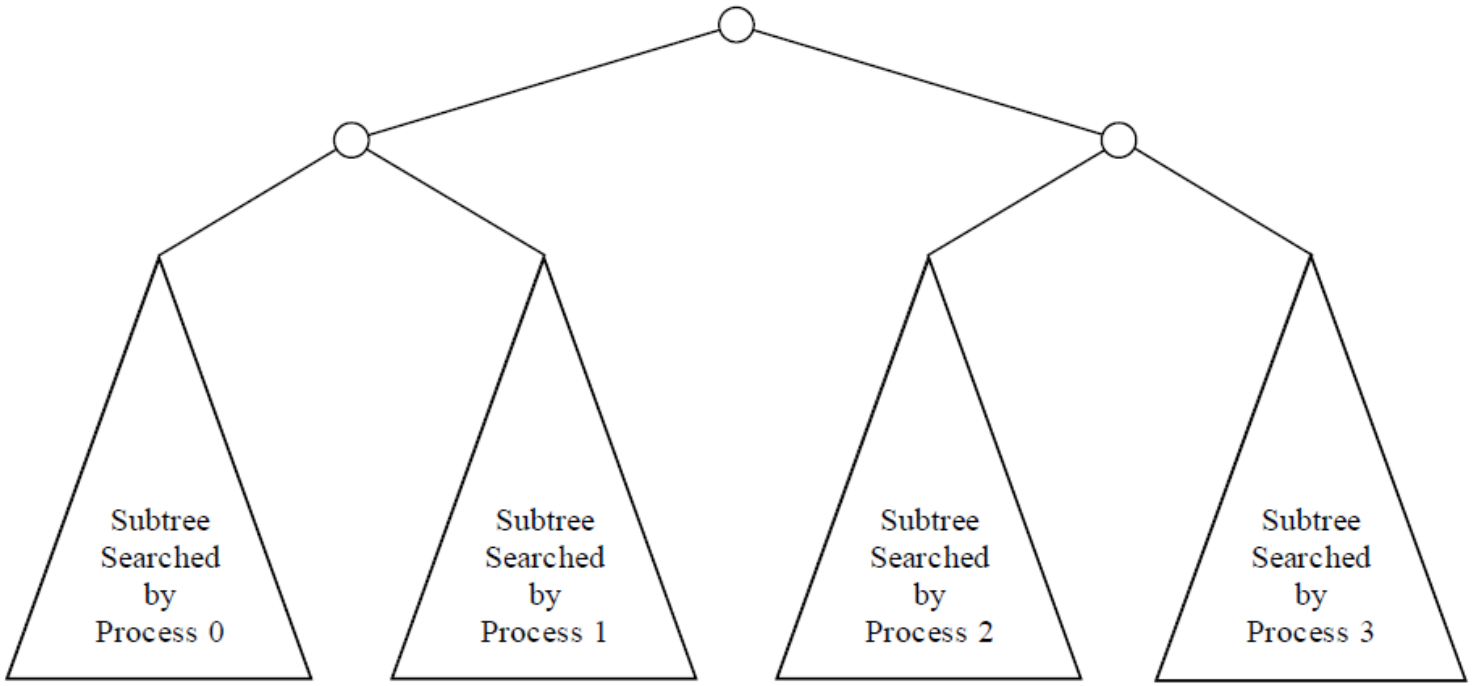
  nodes in the worst case (exponential time)
- Amount of memory required is $\Theta(k)$



Parallel Depth-First Search

- First strategy: give each processor a subtree
- Suppose $p = b^k$
  - A process searches all nodes to depth k
  - It then explores only one of subtrees rooted at level k
  - If d (depth of search) > $2k$, time required by each process to traverse first k levels of state space tree inconsequential

Parallel Backtrack when $p = b^k$

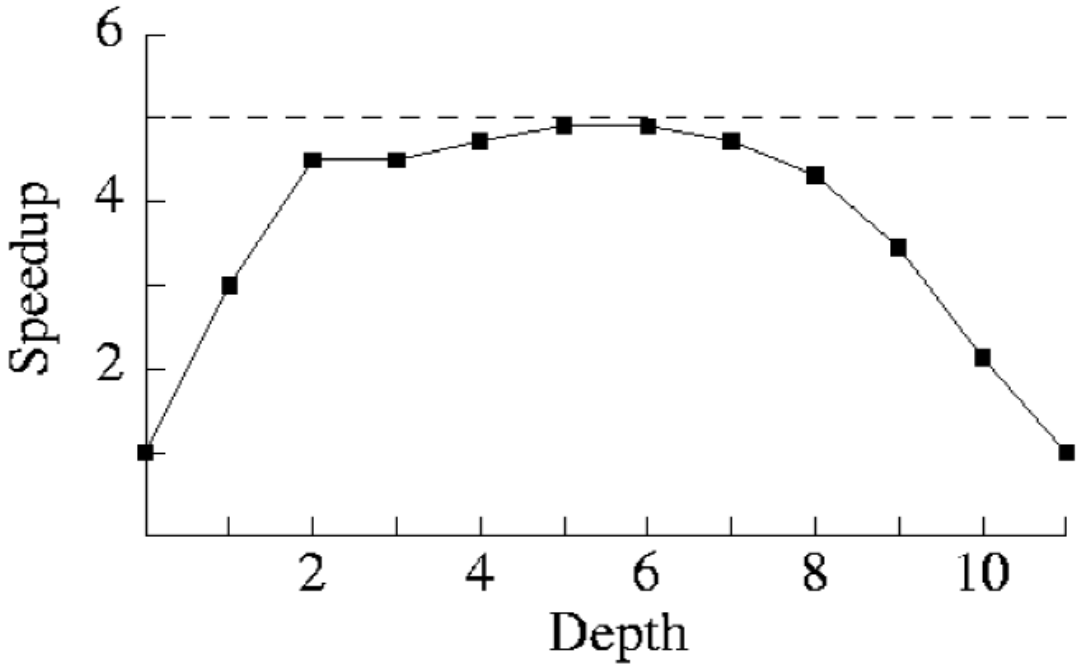


What If $p \neq b^k$?

- A process can perform sequential search to level m of state space tree
- Each process explores its share of the subtrees rooted by nodes at level m
  - As m increases, there are more subtrees to divide among processes, which can make workloads more balanced
  - Increasing m also increases number of redundant computations

Maximum Speedup when $p \neq b^k$

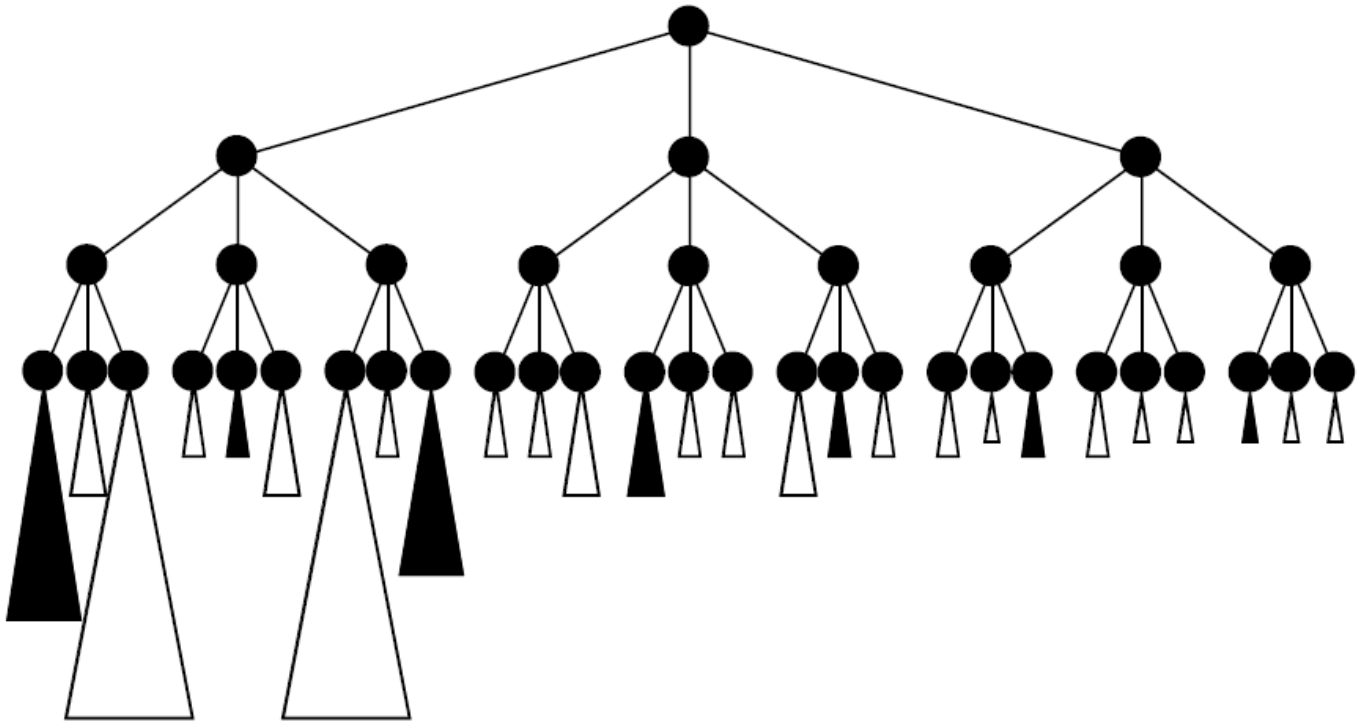
In this example 5 processors are exploring a state space tree with branching factor 3 and depth 10.

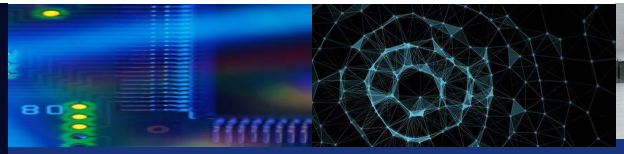


Disadvantage of Allocating One Subtree per Process

- In most cases state space tree is not balanced
- Example: in crossword puzzle problem, some word choices lead to dead ends quicker than others
- Alternative: make sequential search go deeper, so that each process handles many subtrees (cyclic allocation)

Allocating Many Subtrees per Process

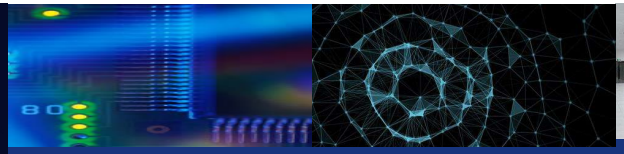




Parallel DFS: Motivation

- **Discrete optimizations are usually NP-hard problems. Does parallelism really help much?**
  - **For many problems, the average-case runtime is polynomial.**
  - **Often, we can find suboptimal solutions in polynomial time.**
  - **Many problems have smaller state spaces but require real-time solutions.**
  - **For some other problems, an improvement in objective function is highly desirable, irrespective of time.**

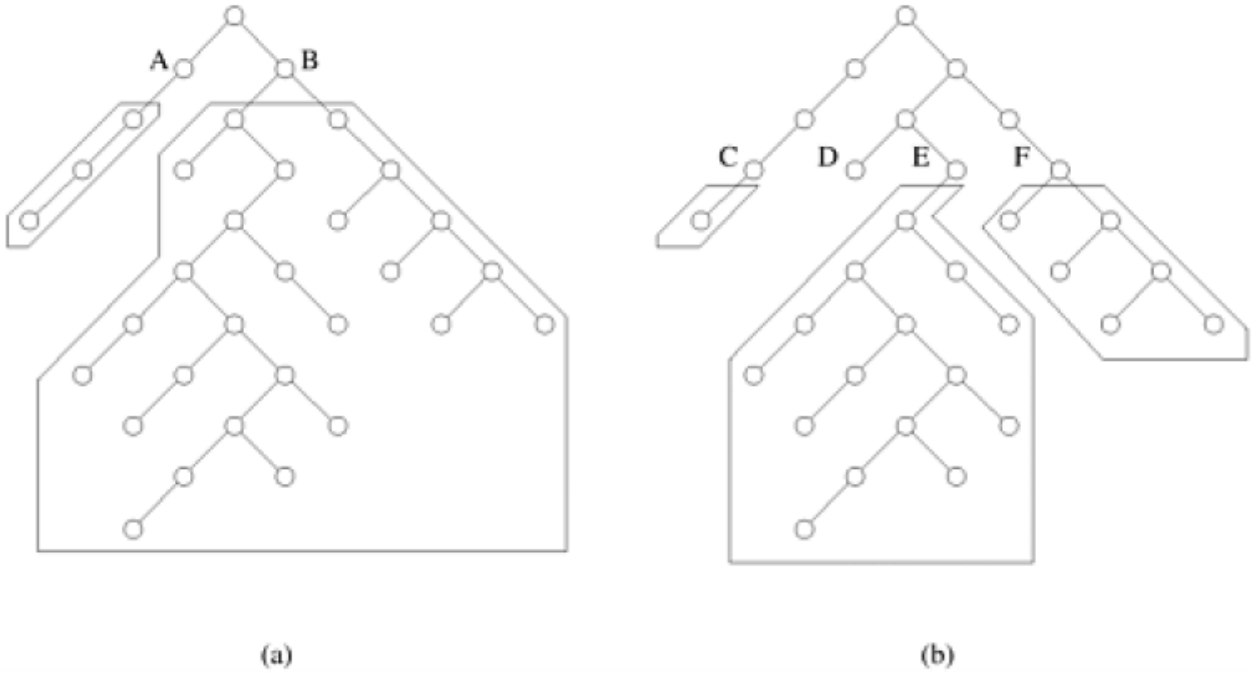
A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003



Parallel Depth-First Search

- How is the search space partitioned across processors?
  - Different subtrees can be searched concurrently.
  - However, subtrees can be very different in size.
  - It is difficult to estimate the size of a subtree rooted at a node.
- Dynamic load balancing is required.

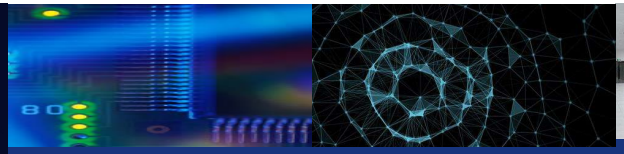
Parallel Depth-First Search



The unstructured nature of tree search and the imbalance resulting from static partitioning.

Parameters in Parallel DFS: Work Splitting

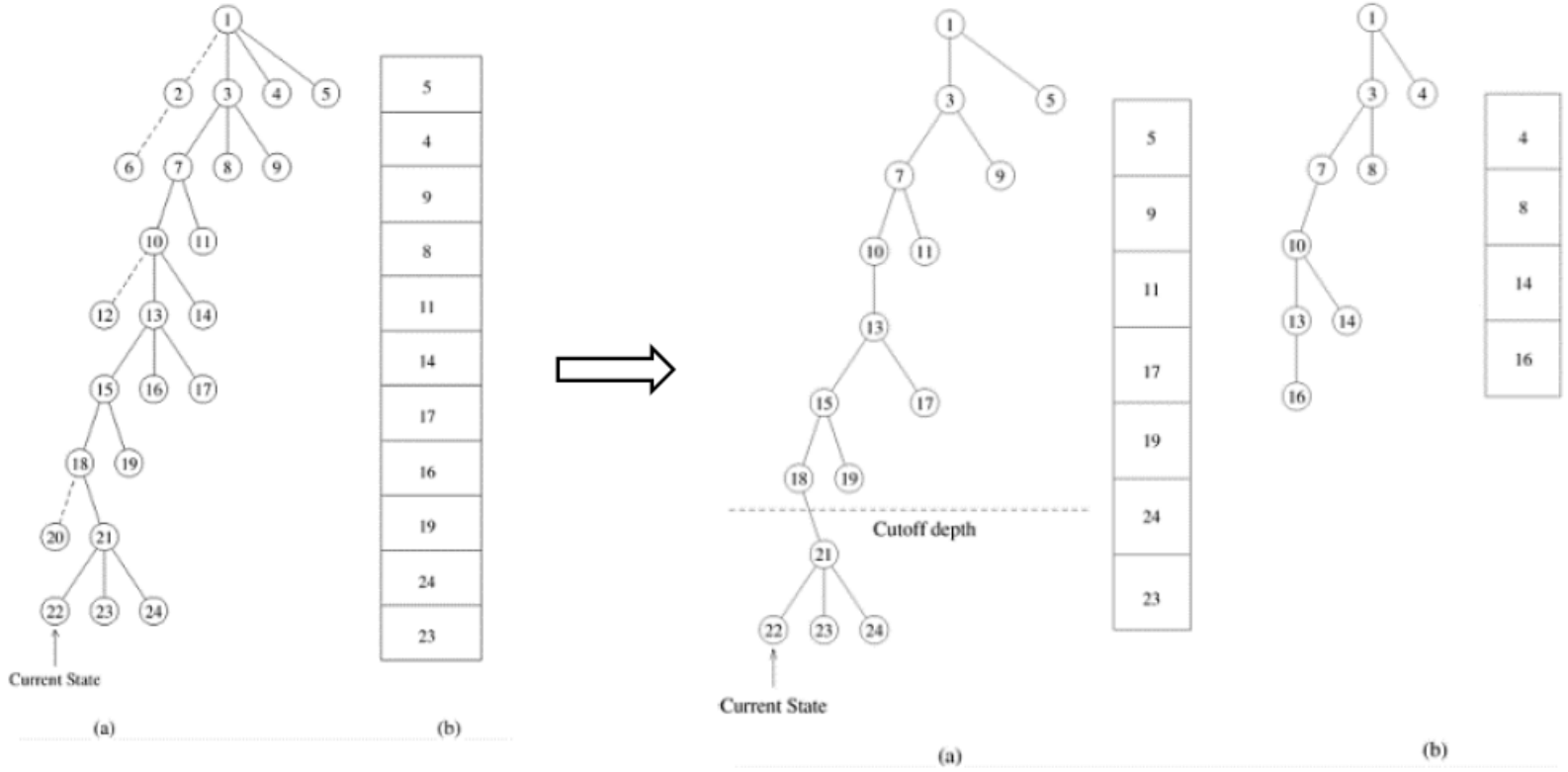
- **Terminologies**
  - *Donor* process: the process that sends work
  - *Recipient* process: the process that requests/receives work
  - *Half-split*: ideally, the stack is split into two equal pieces such that the search space of each stack is the same
  - *Cutoff depth*: to avoid sending very small amounts of work, nodes beyond a specified stack depth are not given away



Parameters in Parallel DFS: Work Splitting

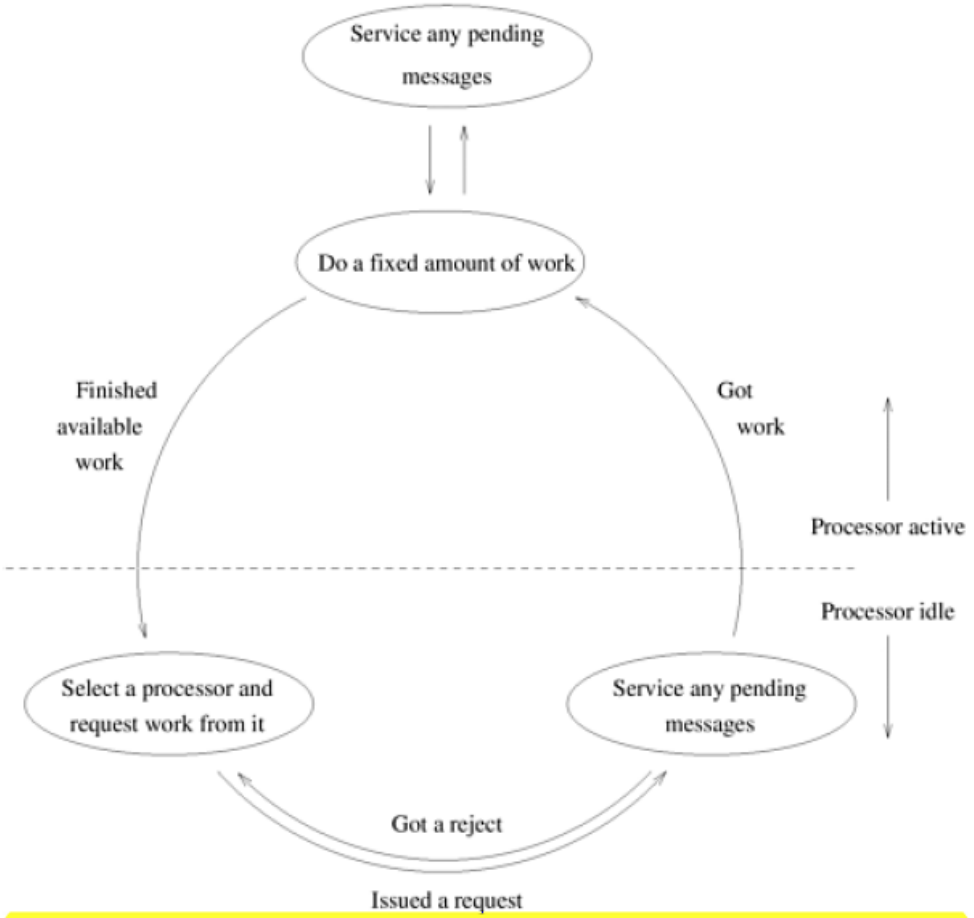
- **Some possible strategies**
  1. **Send nodes near the bottom of the stack**
     - **Works well with uniform search space; has low splitting cost**
  2. **Send nodes near the cutoff depth**
     - **Performs better with a strong heuristic (tries to distribute the parts of the search space likely to contain a solution)**
  3. **Send half the nodes between the bottom and the cutoff depth**
     - **Works well with uniform and irregular search space**

Parameters in Parallel DFS: Work Splitting

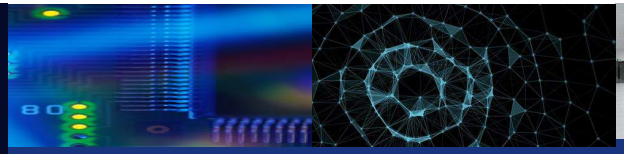


Splitting the DFS tree:
the two subtrees along with their stack representations.

Parallel DFS: Dynamic Load Balancing

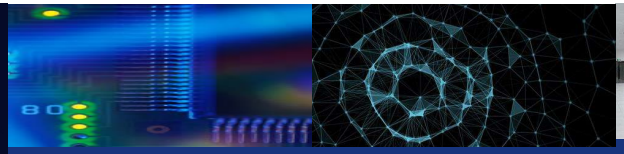


A generic scheme for dynamic load balancing.



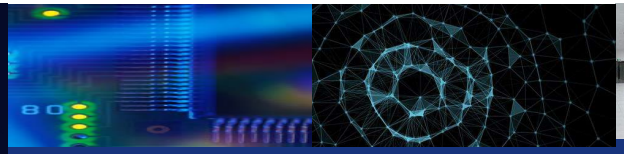
Parallel DFS: Dynamic Load Balancing

- The entire space is assigned to one processor to begin with.
- When a processor runs out of work, it gets more work from another processor.
  - Message passing machines: work requests and responses
  - Shared address space machines: locking and extracting work
- Unexplored states can be conveniently stored as local stacks at processors.
- On reaching final state at a processor, all processors terminate.



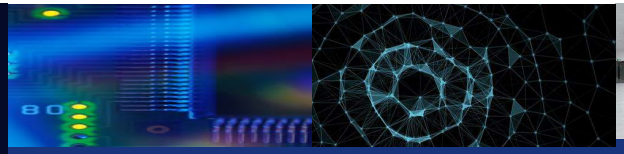
Load-Balancing Schemes

- **Who do you request work from? Note that we would like to distribute work requests evenly, in a global sense.**
- **Asynchronous round robin (ARR)**
  - **Each process maintains a counter and makes requests in a round-robin fashion.**
- **Global round robin (GRR)**
  - **The system maintains a global counter and requests are made in a round-robin fashion, globally.**
- **Random polling (RP)**
  - **Request a randomly selected process for work.**



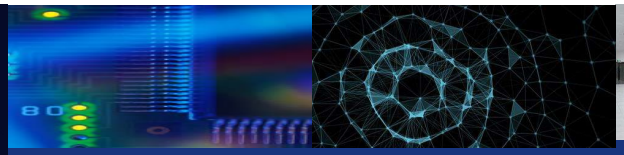
Analyzing DFS

- We can't compute, analytically, the serial work W or parallel time W_p in terms of input size n
- Instead, we quantify total overhead T_o in terms of W to compute scalability.
  - $T_O = pW_p - W$
  - Overhead is due to
    - Communication (requesting and sending work)
    - Idle time (waiting for work)
    - Termination detection
    - Contention for shared resources (e.g., the global counter in GRR)
    - Search overhead factor
  - For dynamic load balancing, idling time is subsumed by communication.
- We must quantify the total number of requests in the system.



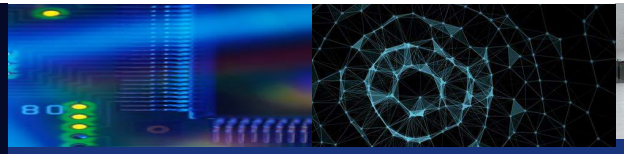
Search Overhead Factor

- The amount of work done by serial and parallel formulations of search algorithms is often different.
- Let W be serial work and pW_P be parallel work. Search overhead factor s is defined as pW_P/W.
- Upper bound on speedup is $p \times 1/s$.
  - ATTENTION: $W/W_P < 1$ is possible (speedup anomalies)



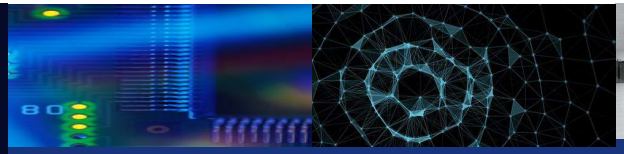
Analyzing DFS: Assumptions

- Search overhead factor = one
- Work at any processor can be partitioned into independent pieces as long as its size exceeds a threshold ε.
- A reasonable work-splitting mechanism is available.
  - If work w at a processor is split into two parts ψw and $(1-\psi)w$, there exists an arbitrarily small constant α $(0 < \alpha \leq 0.5)$, such that $\psi w > \alpha w$ and $(1-\psi)w > \alpha w$.
  - The constant α sets a lower bound on the load imbalance from work splitting.



Analyzing DFS

- If processor P_i initially had work w_i, after a single request by processor P_j and split, neither P_i nor P_j have more than $(1-\alpha)w_i$ work.
- For each load balancing strategy, we define $V(P)$ as the total number of work requests after which each processor receives at least one work request (note that $V(p) \geq p$).
- Assume that the largest piece of work at any point is W.
- After $V(p)$ requests, the maximum work remaining at any processor is less than $(1-\alpha)W$; after $2V(p)$ requests, it is less than $(1-\alpha)^2W$; …
- After $(\log_{1/1(1-\alpha)}(W/\varepsilon))V(p)$ requests, the maximum work remaining at any processor is below a threshold value ε.
- The total number of work requests is $O(V(p) \log W)$.



Analyzing DFS

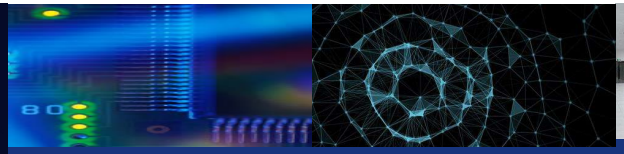
- If t_{comm} is the time required to communicate a piece of work, then the communication overhead T_O is

$$T_O = t_{comm} V(p) \log W$$

The corresponding efficiency E is given by

$$E = \frac{1}{1 + T_o/W}$$

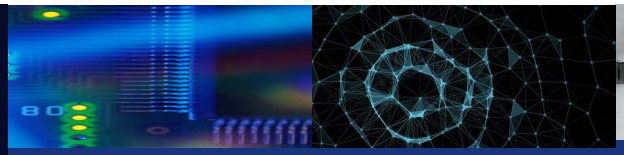
$$= \frac{1}{1 + (t_{comm} V(p) \log W)/W}$$



Analyzing DFS: for Various Schemes

- **Asynchronous Round Robin**
  - $V(p) = O(p^2)$ **in the *worst case*.**
- **Global Round Robin**
  - $V(p) = p$.
- **Random Polling**
  - **Worst case $V(p)$ is unbounded.**
  - **We do *average case* analysis.**

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003



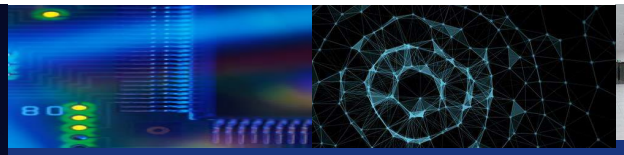
for Random Polling

- Let $F(i,p)$ represent a state in which i of the processors have been requested, and $p–i$ have not.
- Let $f(i,p)$ denote the average number of trials needed to change from state $F(i,p)$ to $F(p,p)$ ($V(p) = f(0,p)$).

$$f(i,p) = \frac{i}{p}(1 + f(i,p)) + \frac{p-i}{p}(1 + f(i+1,p)),$$

$$\frac{p-i}{p} f(i,p) = 1 + \frac{p-i}{p} f(i+1,p),$$

$$f(i,p) = \frac{p}{p-i} + f(i+1,p).$$



for Random Polling

- **We have:**

$$f(0,p) = p \times \sum_{i=0}^{p-1} \frac{1}{p-i},$$

$$= p \times \sum_{i=1}^{p} \frac{1}{i},$$

$$= p \times H_p,$$

H_p ← harmonic number

As p becomes large, $H_p \simeq 1.69 \ln p$. Thus, $V(p) = O(p \log p)$.

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Analysis of Load-Balancing Schemes

- **If** $t_{comm} = O(1)$**, we have** $T_0 = O(V(p) log\ W)$**.**
- **Asynchronous Round Robin: Since** $V(p) = O(p^2)$**,** $T_0 = O(p^2 \log w)$**. It follows that:**

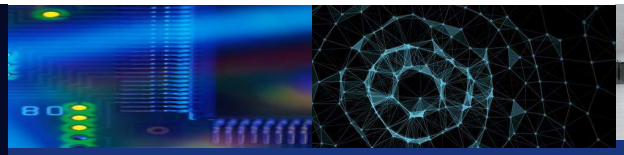
Balance overhead and problem size

$$W = O(p^2 \log(p^2 \log W)),$$

$$= O(p^2 \log p + p^2 \log \log W)$$

$$= O(p^2 \log p)$$

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003



Analysis of Load-Balancing Schemes

- **Global Round Robin: Since** $V(p) = O(p)$, $T_0 = O(p \log W)$. **It follows that** $W = O(p \log p)$.

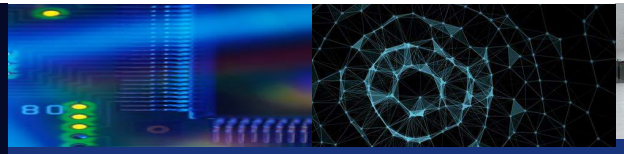
  However, there is contention here! The global counter must be incremented $O(p \log W)$ **times in** $O(W/p)$ **time.**

  From this, we have: $$\frac{W}{p} = O(p \log W)$$

  and $W = O(p^2 \log p)$.

  The worse of these two expressions, $W = O(p^2 \log p)$ **is the isoefficiency.**

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003



Analysis of Load-Balancing Schemes

- **Random Polling: We have** $V(p) = O(p \log p)$,

  $T_o = O(p \log p \log W)$

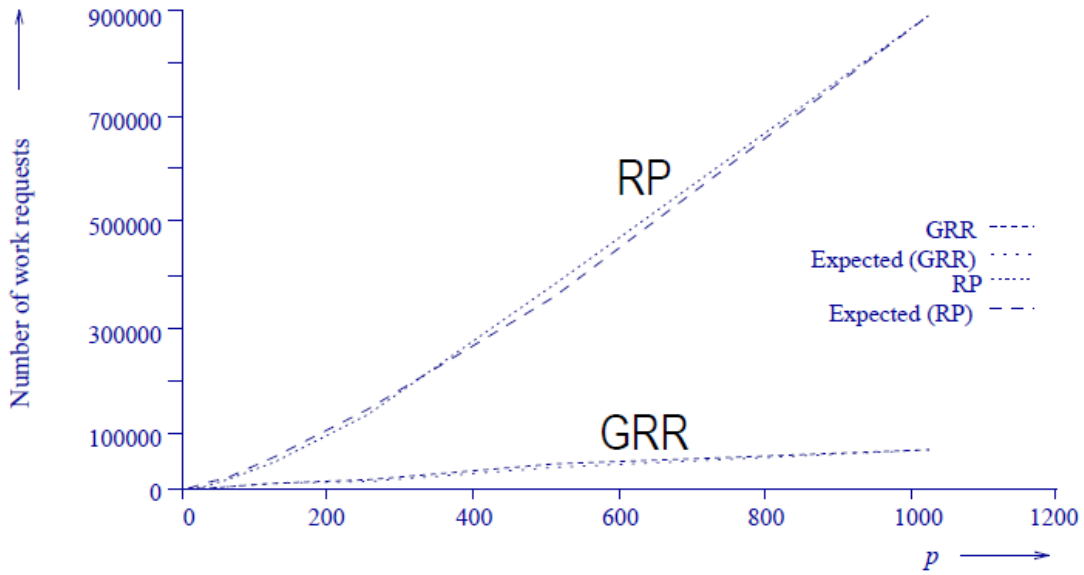
  Therefore $W = O(p \log^2 p)$.

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Analysis of Load-Balancing Schemes: Conclusions

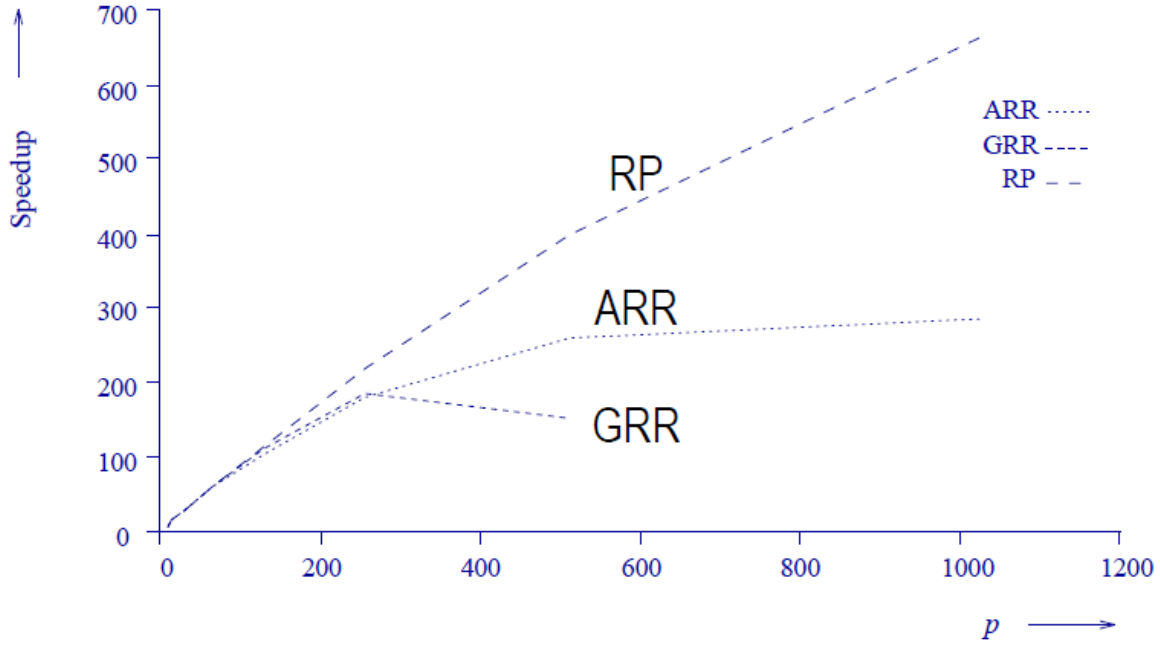
- **Asynchronous round robin has poor performance because it makes a large number of work requests.**
- **Global round robin has poor performance because of contention at counter, although it makes the least number of requests.**
- **Random polling strikes a desirable compromise.**

Experimental Validation: Satisfiability Problem



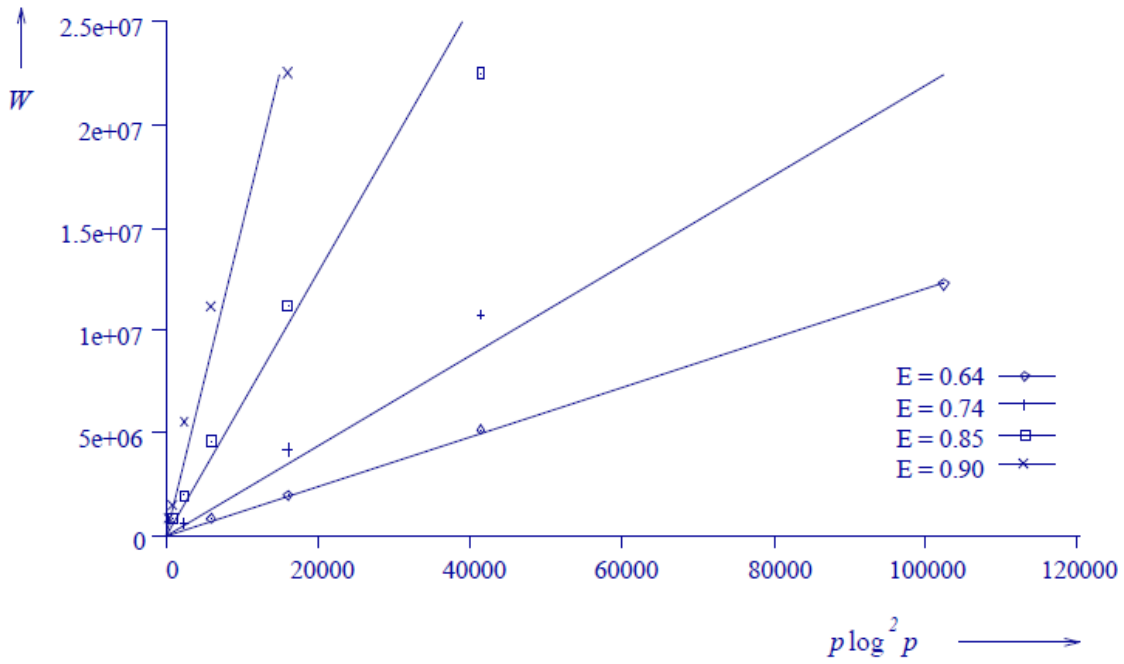
Number of work requests generated for RP and GRR and their expected values ($O(p \log^2 p)$ and $O(p \log p)$ respectively).

Experimental Validation: Satisfiability Problem



Speedups of parallel DFS using ARR, GRR and RP load-balancing schemes.

Experimental Validation: Satisfiability Problem



Experimental isoefficiency curves for RP for different efficiencies.

华容道


三国华容道
黄忠
曹操
赵云
关羽
张飞
马超
卒
卒
卒
卒

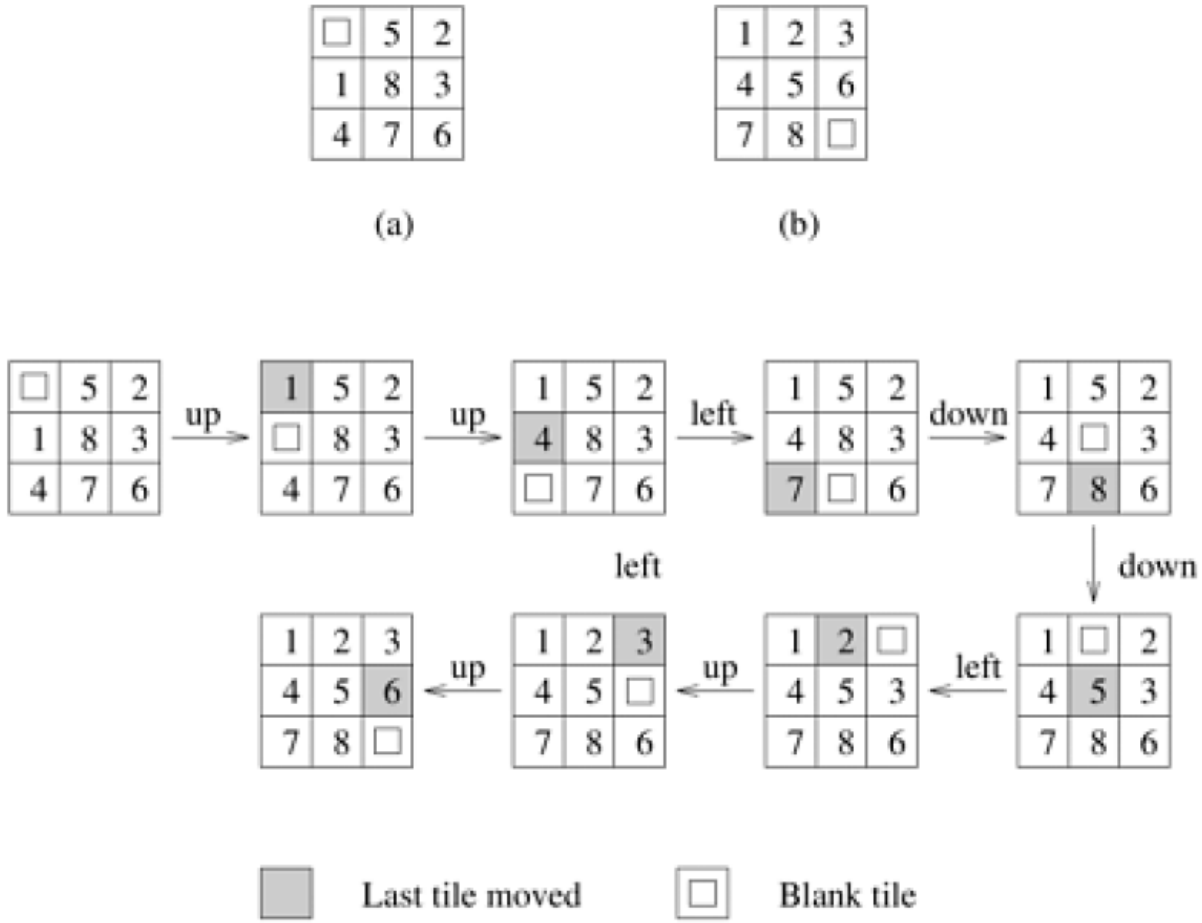

吉妮兔®
Ginirabbit
数字华容道
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15

Best-First Search: 8-puzzle

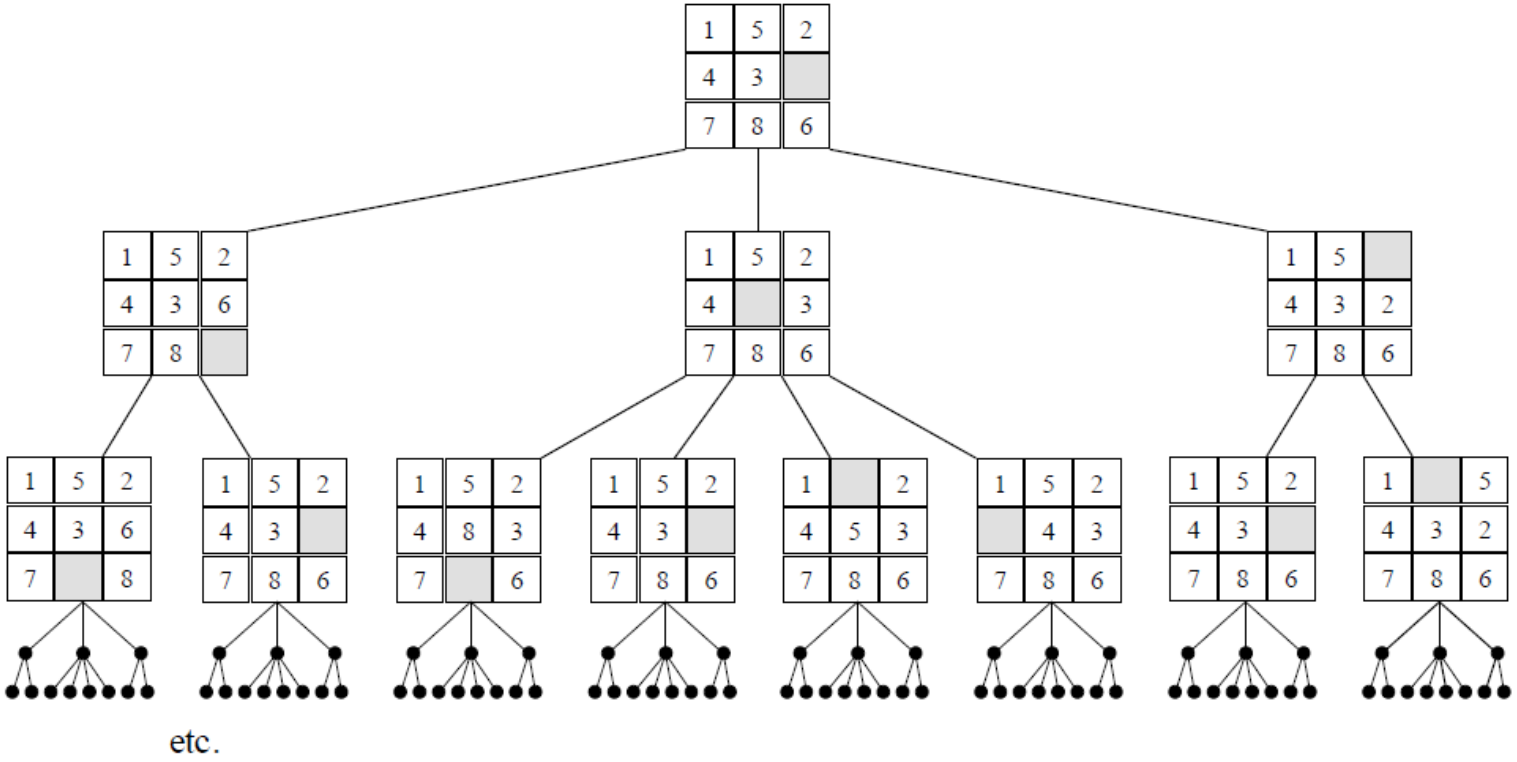
1	2	3
4	5	6
7	8	Hole

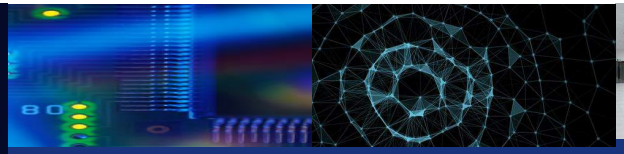
This is the solution state. Tiles slide up, down, or sideways into hole.

Example: Solve 8-Puzzle Problem



State Space Tree Represents Possible Moves





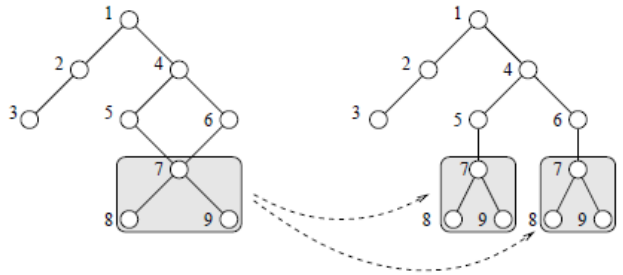
Example: 8-Puzzle Problem

- The 8-puzzle problem consists of a 3×3 grid containing eight tiles, numbered one through eight.
- One of the grid segments (called the “blank”) is empty. A tile can be moved into the blank position from a position adjacent to it, thus creating a blank in the tile's original position.
- The goal is to move from a given initial position to the final position in a minimum number of moves.

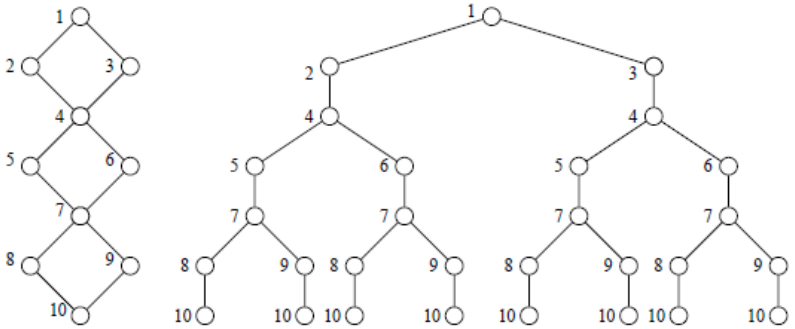
Search Space

- Is the search space a tree or a graph?
  - The space of an 8-puzzle is a graph.
- This has important implications for search since unfolding a graph into a tree can have significant overheads.

Unfolded Search Tree



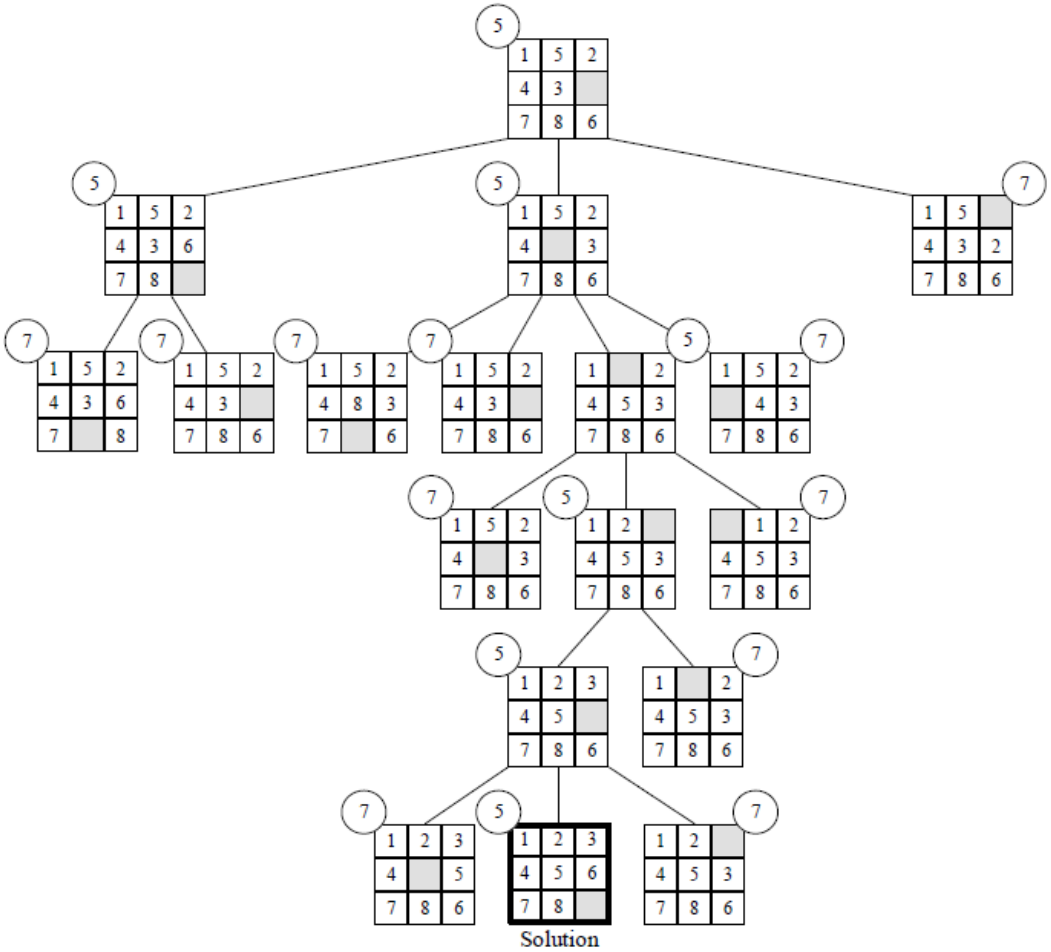
(a)



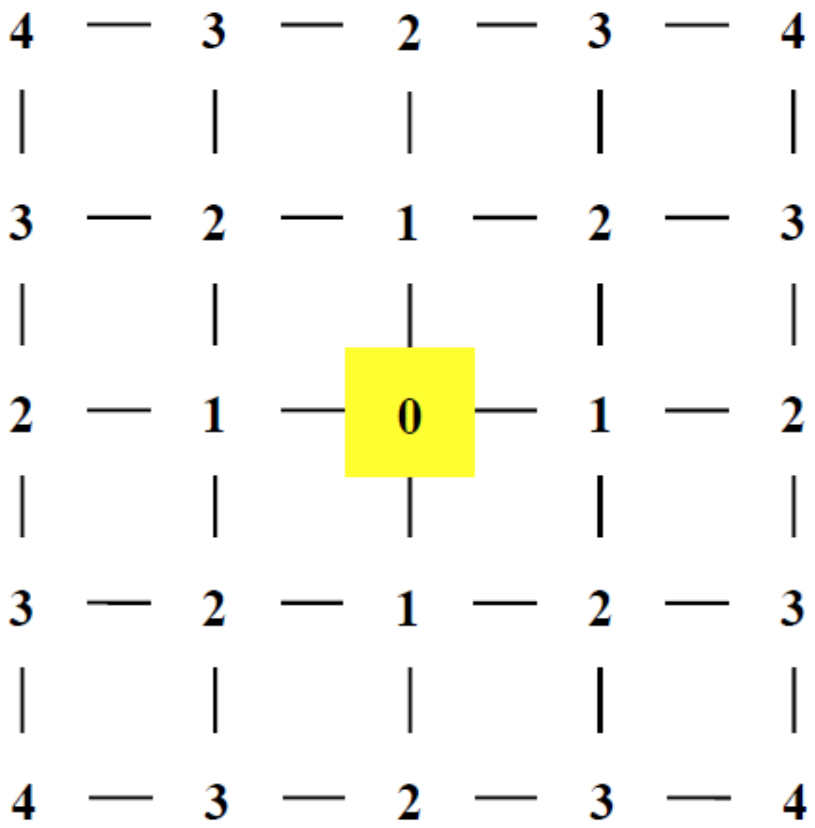
(b)

Two examples of unfolding a graph into a tree.

Best-First Search of 8-puzzle



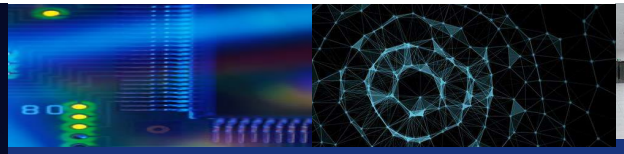
Manhattan Distance



Manhattan distance from the yellow intersection.

A Lower Bound Function

- A lower bound on number of moves needed to solve puzzle is sum of Manhattan distance of each tile's current position from its correct position
- Depth of node in state space tree indicates number of moves made so far
- Adding two values gives lower bound on number of moves needed for any solution, given moves made so far
- We always search from node having smallest value of this function (best-first search)

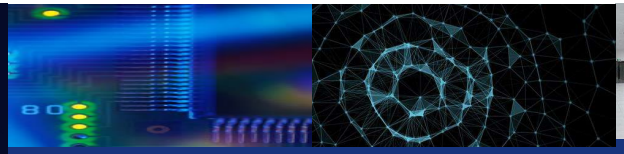


Pseudocode: Sequential Algorithm

```
Intialize (q)
Insert (q, initial)
repeat
    u ← Delete_Min (q)
    if u is a solution then
            Print_solution (u)
            Halt
    else
            for i ← 1 to Possible_Constraints (u) do
                    Add constraint i to u, creating v
                    Insert (q, v)
            endfor
    endif
forever
```

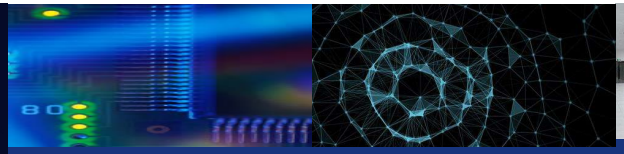
Time and Space Complexity

- In worst case, lower bound function causes function to perform breadth-first search
- Suppose branching factor is b and optimum solution is at depth k of state space tree
- Worst-case time complexity is $\Theta(b^k)$
- On average, b nodes inserted into priority queue every time a node is deleted
- Worst-case space complexity is $\Theta(b^k)$
- Memory limitations often put an upper bound on the size of the problem that can be solved



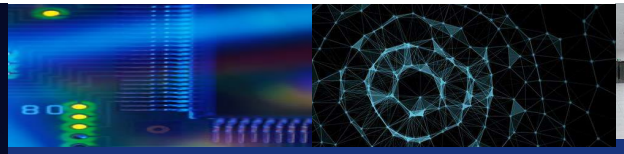
Parallel Best-First Search

- We will develop a parallel algorithm suitable for implementation on a multicomputer or distributed multiprocessor
- Conflicting goals
  - Want to maximize ratio of local to non-local memory references
  - Want to ensure processors searching worthwhile portions of state space tree 重要的，值得



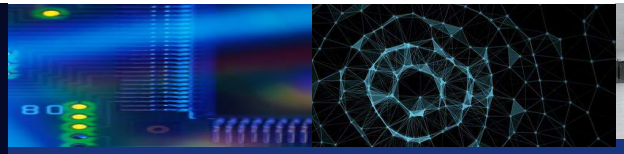
Single Priority Queue

- Maintaining a single priority queue not a good idea
- Communication overhead too great
- Accessing queue is a performance bottleneck
- Does not allow problem size to scale with number of processors



Multiple Priority Queues

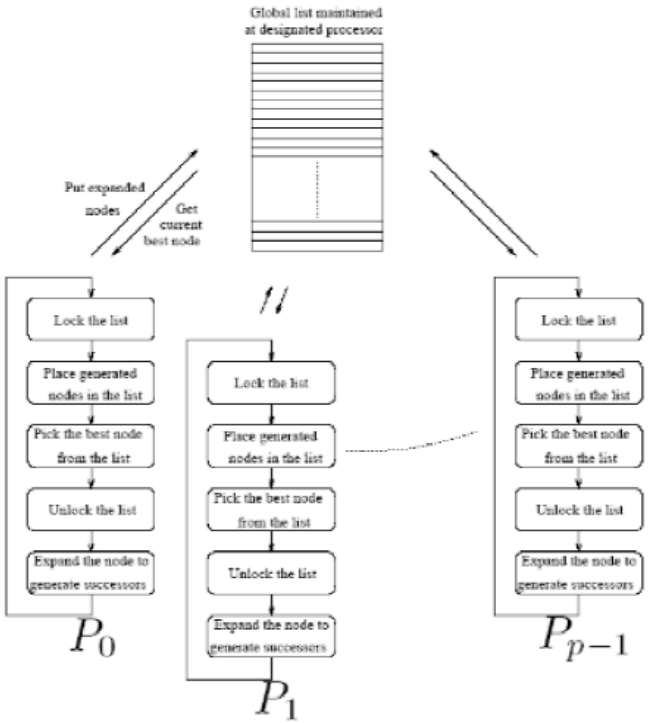
- Each process maintains separate priority queue of unexamined subproblems
- Each process retrieves subproblem with smallest lower bound to continue search
- Occasionally processes send unexamined subproblems to other processes



Parallel Best-First Search

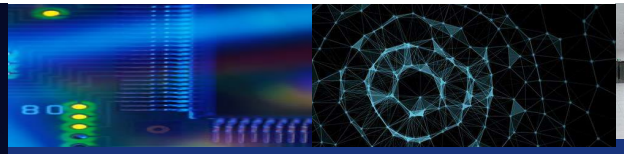
- The core data structure is the open list (typically implemented as a priority queue).
- Each processor locks this queue, extracts the best node, unlocks it.
- Successors of the node are generated, their heuristic functions estimated, and the nodes inserted into the open list as necessary after appropriate locking.
- Termination signaled when we find a solution whose cost is better than the best heuristic value in the open list.
- Since we expand more than one node at a time, we may expand nodes that would not be expanded by a sequential algorithm.

Parallel Best-First Search: Centralized Strategy



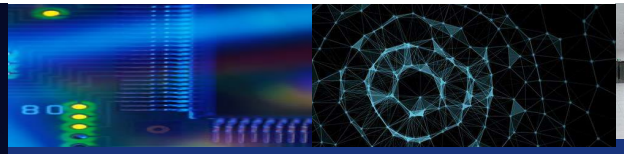
A general schematic for parallel best-first search using a centralized strategy. The locking operation is used here to serialize queue access by various processors.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003



Parallel Best-First Search

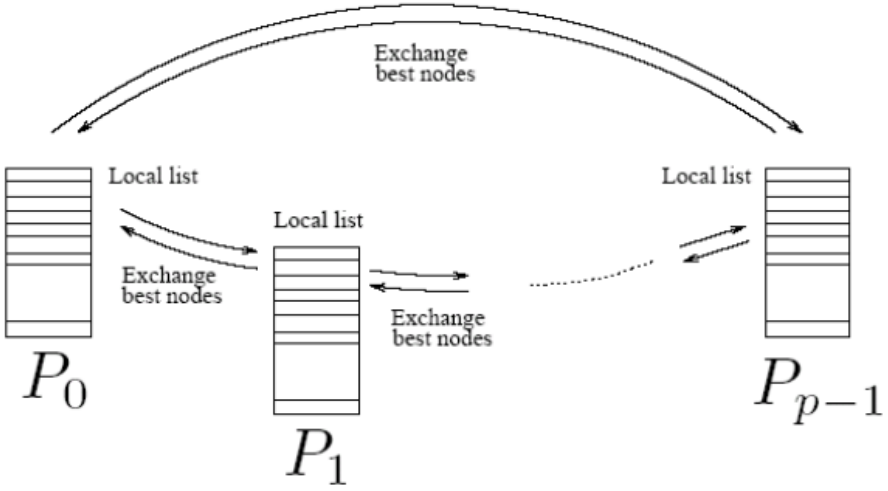
- The open list is a point of contention.
- Let t_{exp} be the average time to expand a single node, and t_{access} be the average time to access the *open* list for a single-node expansion.
- If there are n nodes to be expanded by both the sequential and parallel formulations (assuming that they do an equal amount of work), then the sequential run time is given by $n(t_{access} + t_{exp})$.
- The parallel run time will be at least nt_{access}.
- Upper bound on the speedup is $(t_{access} + t_{exp})/t_{access}$



Parallel Best-First Search

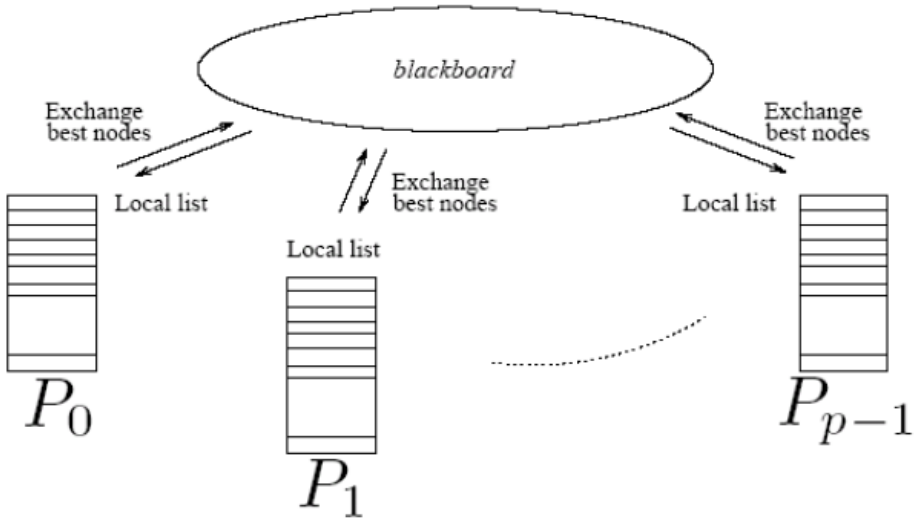
- Avoid contention by having multiple open lists.
- Initially, the search space is statically divided across these open lists.
- Processors concurrently operate on these open lists.
- Since the heuristic values of nodes in these lists may diverge significantly, we must periodically balance the quality of nodes in each list.
- A number of balancing strategies based on (i) *random*, (ii) *ring*, or (iii) *blackboard* communications are possible.

Parallel Best-First Search: Ring Communication Strategy

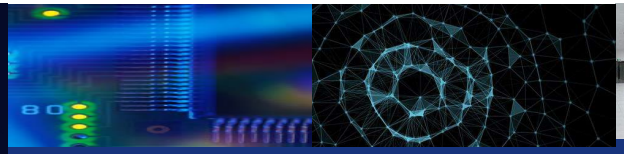


A message-passing implementation of parallel best-first search using the ring communication strategy.

Parallel Best-First Search: Blackboard Communication Strategy



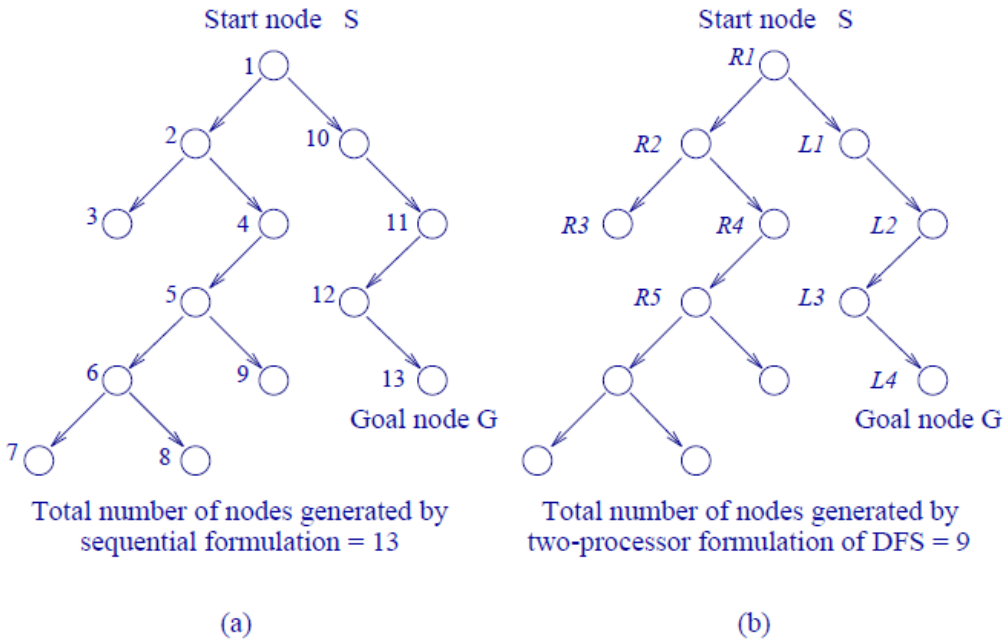
An implementation of parallel best-first search using the blackboard communication strategy.



Speedup Anomalies in Parallel Search

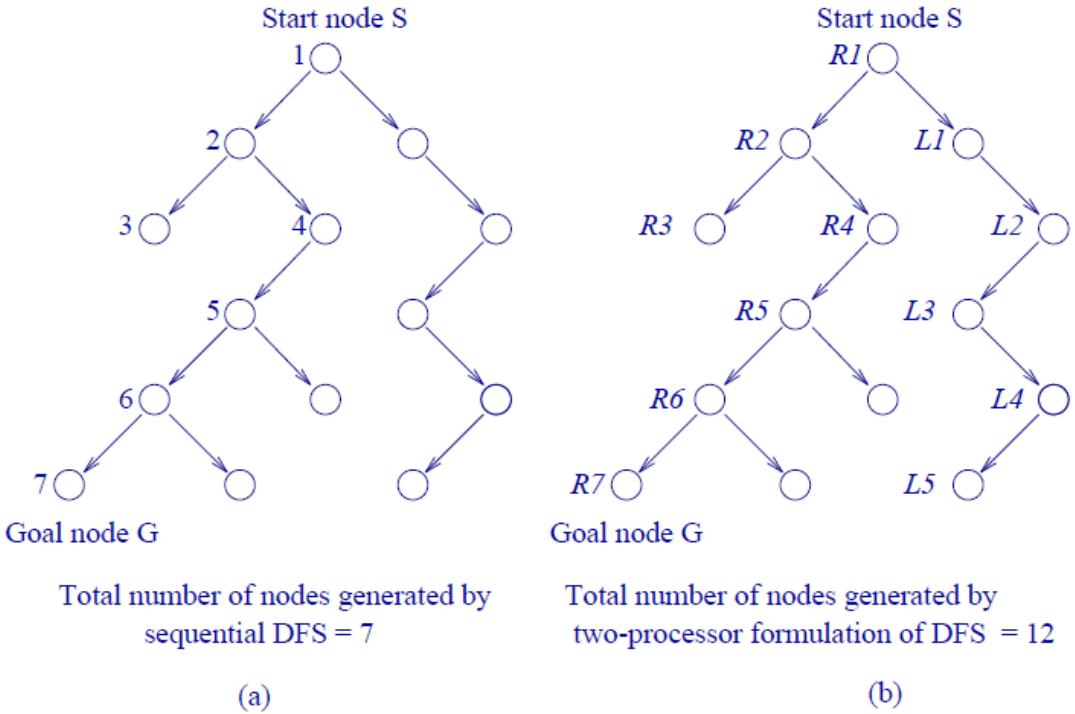
- Since the search space explored by processors is determined dynamically at runtime, the actual work might vary significantly.
- Executions yielding speedups greater than p by using p processors are referred to as *acceleration anomalies*. Speedups of less than p using p processors are called *deceleration anomalies*.
- Speedup anomalies also manifest themselves in best-first search algorithms.
- If the heuristic function is good, the work done in parallel best-first search is typically more than that in its serial counterpart.

Speedup Anomalies in Parallel Search

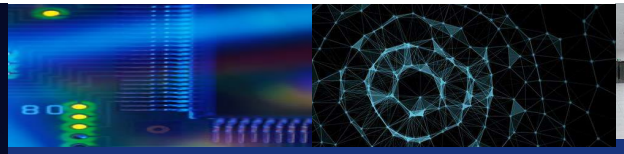


The difference in number of nodes searched by sequential and parallel formulations of DFS. For this example, parallel DFS reaches a goal node after searching fewer nodes than sequential DFS.

Speedup Anomalies in Parallel Search

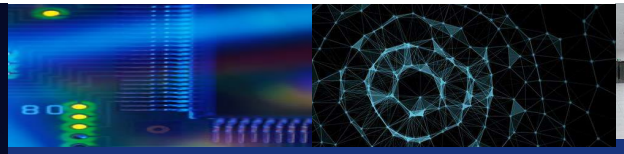


A parallel DFS formulation that searches more nodes than its sequential counterpart



Summary

- Parallel depth-first search
  - Load balancing schemes: ARR, GRR, RP
  - Scalability analysis
- Parallel best-first search
  - Centralized strategy
  - Communication strategies: random, ring, blackboard
- Speedup anomalies



Thank You !