Search Algorithms for Discrete Optimization Problems

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text "Introduction to Parallel Computing",
Addison Wesley, 2003.

Topic Overview

- Discrete Optimization – Basics
- Sequential Search Algorithms
- Parallel Depth-First Search
- Parallel Best-First Search
- Speedup Anomalies in Parallel Search Algorithms

Discrete Optimization – Basics

- Discrete optimization forms a class of computationally expensive problems of significant theoretical and practical interest.

- Search algorithms systematically search the space of possible solutions subject to constraints.

Definitions

- A *discrete optimization problem* can be expressed as a tuple (S, f). The set S is a finite or countably infinite set of all solutions that satisfy specified constraints.

- The function f is the cost function that maps each element in set S onto the set of real numbers R.

- The objective of a DOP is to find a feasible solution x_{opt}, such that $f(x_{opt}) \leq f(x)$ for all $x \in S$.

- A number of diverse problems such as VLSI layouts, robot motion planning, test pattern generation, and facility location can be formulated as DOPs.

Discrete Optimization: Example

- In the 0/1 integer-linear-programming problem, we are given an $m \times n$ matrix A, an $m \times 1$ vector b, and an $n \times 1$ vector c.

- The objective is to determine an $n \times 1$ vector $\overline{x}$ whose elements can take on only the value 0 or 1.

- The vector must satisfy the constraint

$$A\overline{x} \geq b$$

  and the function

$$f(\overline{x}) = c^T \overline{x}$$

  must be minimized.

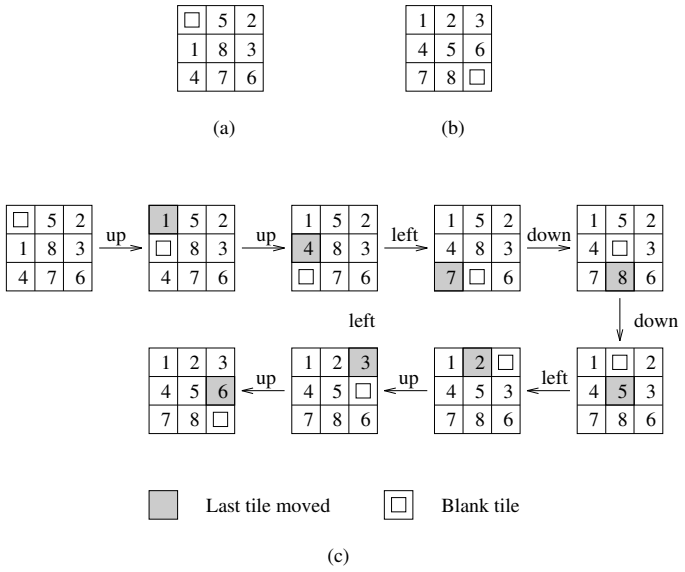
Discrete Optimization: Example

- 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.

Discrete Optimization: Example



An 8-puzzle problem instance: (a) initial configuration; (b) final configuration; and (c) a sequence of moves leading from the initial to the final configuration.

Discrete Optimization Basics

- The feasible space S is typically very large.
- For this reason, a DOP can be reformulated as the problem of finding a minimum-cost path in a graph from a designated initial node to one of several possible goal nodes.
- Each element x in S can be viewed as a path from the initial node to one of the goal nodes.
- This graph is called a *state space*.

Discrete Optimization Basics

- Often, it is possible to estimate the cost to reach the goal state from an intermediate state.
- This estimate, called a *heuristic estimate*, can be effective in guiding search to the solution.
- If the estimate is guaranteed to be an underestimate, the heuristic is called an *admissible heuristic*.
- Admissible heuristics have desirable properties in terms of optimality of solution (as we shall see later).

Discrete Optimization: Example

An admissible heuristic for 8-puzzle is as follows:

- Assume that each position in the 8-puzzle grid is represented as a pair.
- The distance between positions (i, j) and (k, l) is defined as $|i$ $k| + |j \quad l|$. This distance is called the Manhattan distance.
- The sum of the Manhattan distances between the initial and final positions of all tiles is an admissible heuristic.

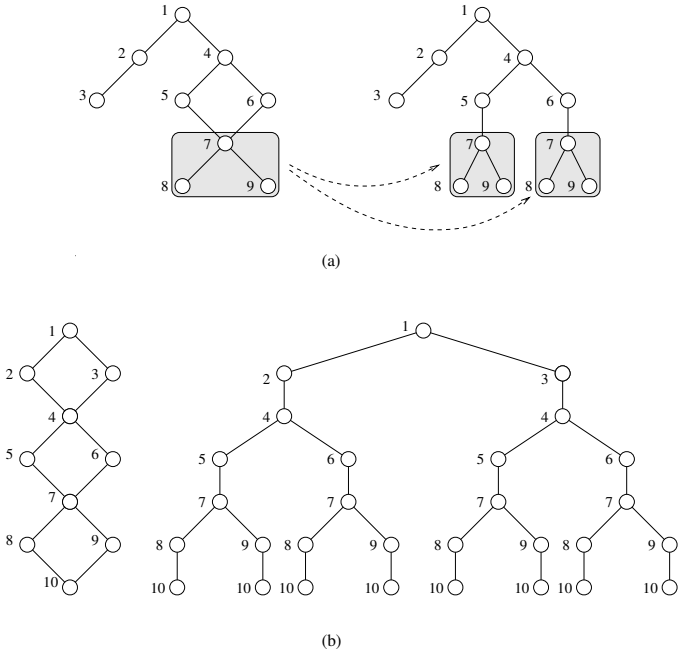
Parallel Discrete Optimization: Motivation

- DOPs are generally 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.

Sequential Search Algorithms

- Is the search space a tree or a graph?
- The space of a 0/1 integer program is a tree, while that of an 8-puzzle is a graph.
- This has important implications for search since unfolding a graph into a tree can have significant overheads.

Sequential Search Algorithms

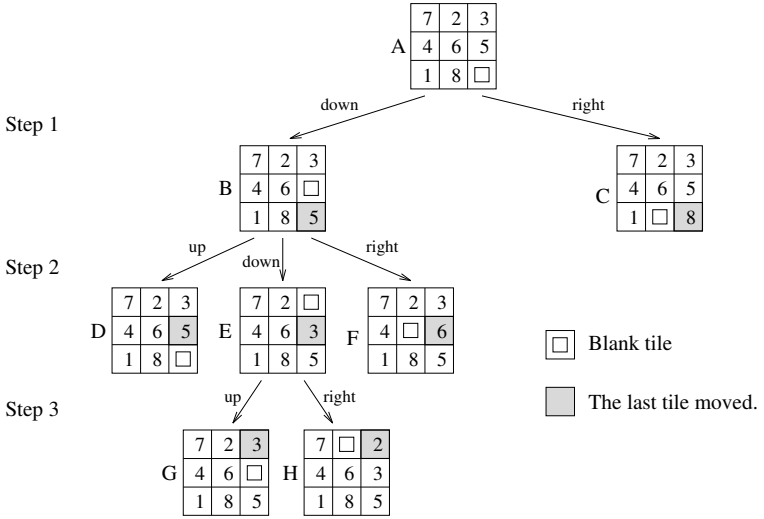


Two examples of unfolding a graph into a tree.

Depth-First Search Algorithms

- Applies to search spaces that are trees.
- DFS begins by expanding the initial node and generating its successors. In each subsequent step, DFS expands one of the most recently generated nodes.
- If there exists no success, DFS backtracks to the parent and explores an alternate child.
- Often, successors of a node are ordered based on their likelihood of reaching a solution. This is called directed DFS.
- The main advantage of DFS is that its storage requirement is linear in the depth of the state space being searched.

Depth-First Search Algorithms



States resulting from the first three steps of depth-first search applied to an instance of the 8-puzzle.

DFS Algorithms: Simple Backtracking

- Simple backtracking performs DFS until it finds the first feasible solution and terminates.
- Not guaranteed to find a minimum-cost solution.
- Uses no heuristic information to order the successors of an expanded node.
- Ordered backtracking uses heuristics to order the successors of an expanded node.

Depth-First Branch-and-Bound (DFBB)

- DFS technique in which upon finding a solution, the algorithm updates current best solution.
- DFBB does not explore paths that ae guaranteed to lead to solutions worse than current best solution.
- On termination, the current best solution is a globally optimal solution.

Iterative Deepening Search

- Often, the solution may exist close to the root, but on an alternate branch.
- Simple backtracking might explore a large space before finding this.
- Iterative deepening sets a depth bound on the space it searches (using DFS).
- If no solution is found, the bound is increased and the process repeated.

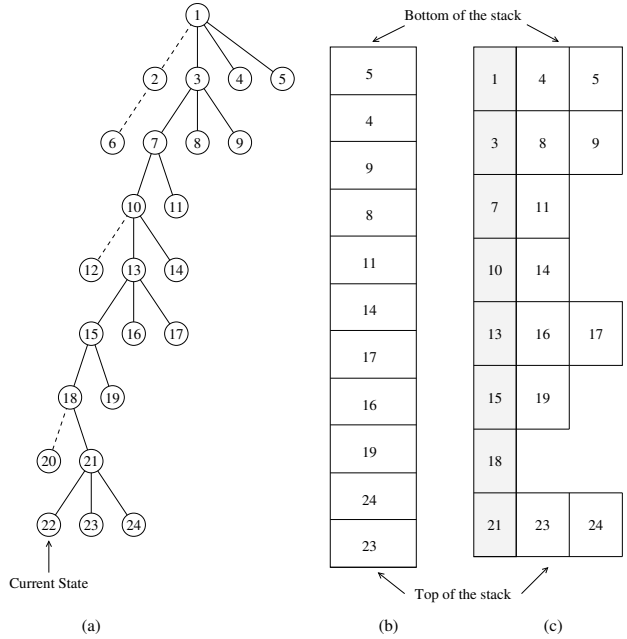
Iterative Deepening A* (IDA*)

- Uses a bound on the cost of the path as opposed to the depth.
- IDA* defines a function for node x in the search space as $l(x) = g(x) + h(x)$. Here, $g(x)$ is the cost of getting to the node and $h(x)$ is a heuristic estimate of the cost of getting from the node to the solution.
- At each failed step, the cost bound is incremented to that of the node that exceeded the prior cost bound by the least amount.
- If the heuristic h is admissible, the solution found by IDA* is optimal.

DFS Storage Requirements and Data Structures

- At each step of DFS, untried alternatives must be stored for backtracking.
- If m is the amount of storage required to store a state, and d is the maximum depth, then the total space requirement of the DFS algorithm is $O(md)$.
- The state-space tree searched by parallel DFS can be efficiently represented as a stack.
- Memory requirement of the stack is linear in depth of tree.

DFS Storage Requirements and Data Structures



Representing a DFS tree: (a) the DFS tree; Successor nodes shown with dashed lines have already been explored; (b) the stack storing untried alternatives only; and (c) the stack storing untried alternatives along with their parent. The shaded blocks represent the parent state and the block to the right represents successor states that have not been explored.

Best-First Search (BFS) Algorithms

- BFS algorithms use a heuristic to guide search.
- The core data structure is a list, called Open list, that stores unexplored nodes sorted on their heuristic estimates.
- The best node is selected from the list, expanded, and its offspring are inserted at the right position.
- If the heuristic is admissible, the BFS finds the optimal solution.

Best-First Search (BFS) Algorithms

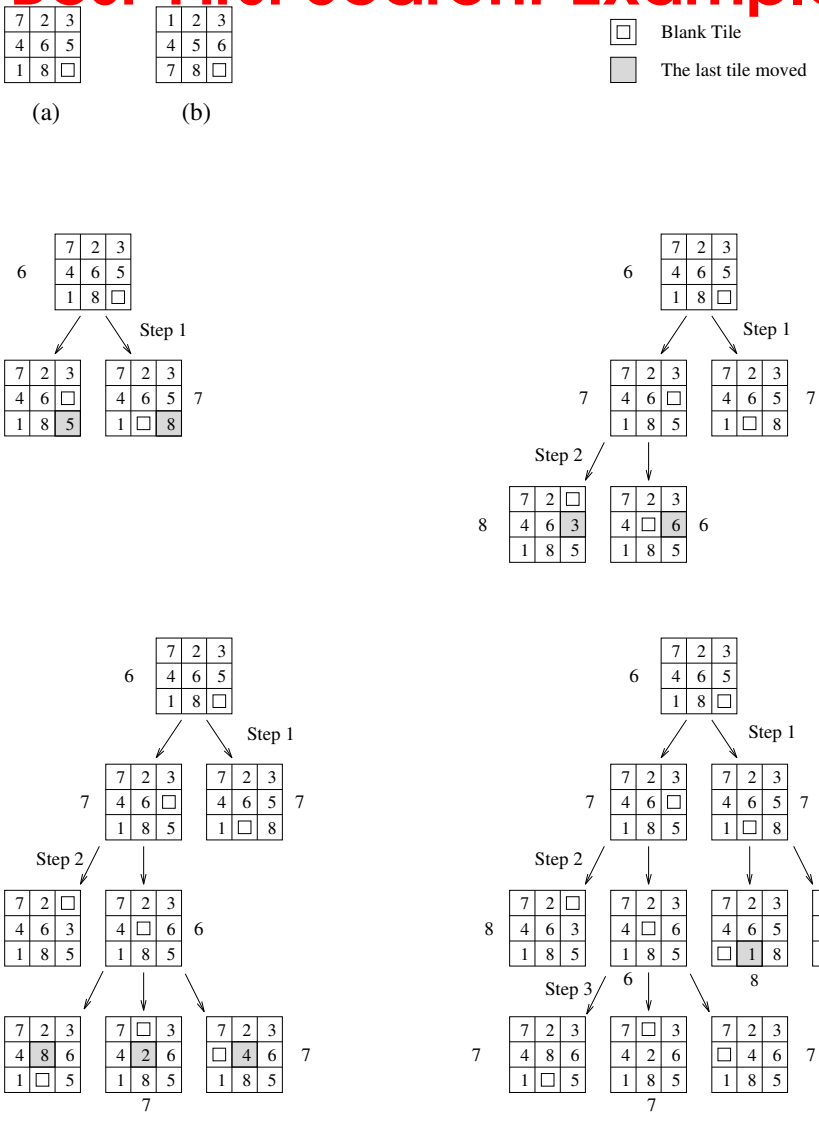
- BFS of graphs must be slightly modified to account for multiple paths to the same node.
- A closed list stores all the nodes that have been previously seen.
- If a newly expanded node exists in the open or closed lists with better heuristic value, the node is not inserted into the open list.

The A* Algorithm

- A BFS technique that uses admissible heuristics.
- Defines function $l(x)$ for each node x as $g(x) + h(x)$.
- Here, $g(x)$ is the cost of getting to node x and $h(x)$ is an admissible heuristic estimate of getting from node x to the solution.
- The open list is sorted on $l(x)$.

The space requirement of BFS is exponential in depth!

Best-First Search: Example



Applying best-first search to the 8-puzzle: (a) initial configuration; (b) final configuration; and (c) states resulting from the first four steps of best-first search. Each state is labeled with its h-value (that is, the Manhattan distance from the state to the final state).

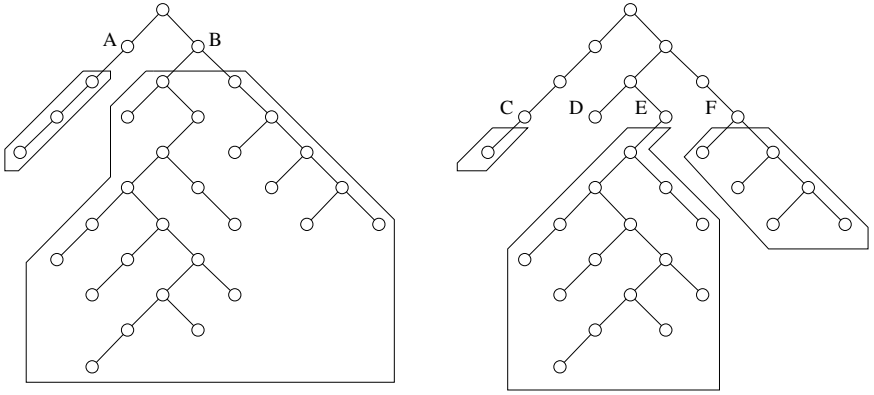
Search Overhead Factor

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

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



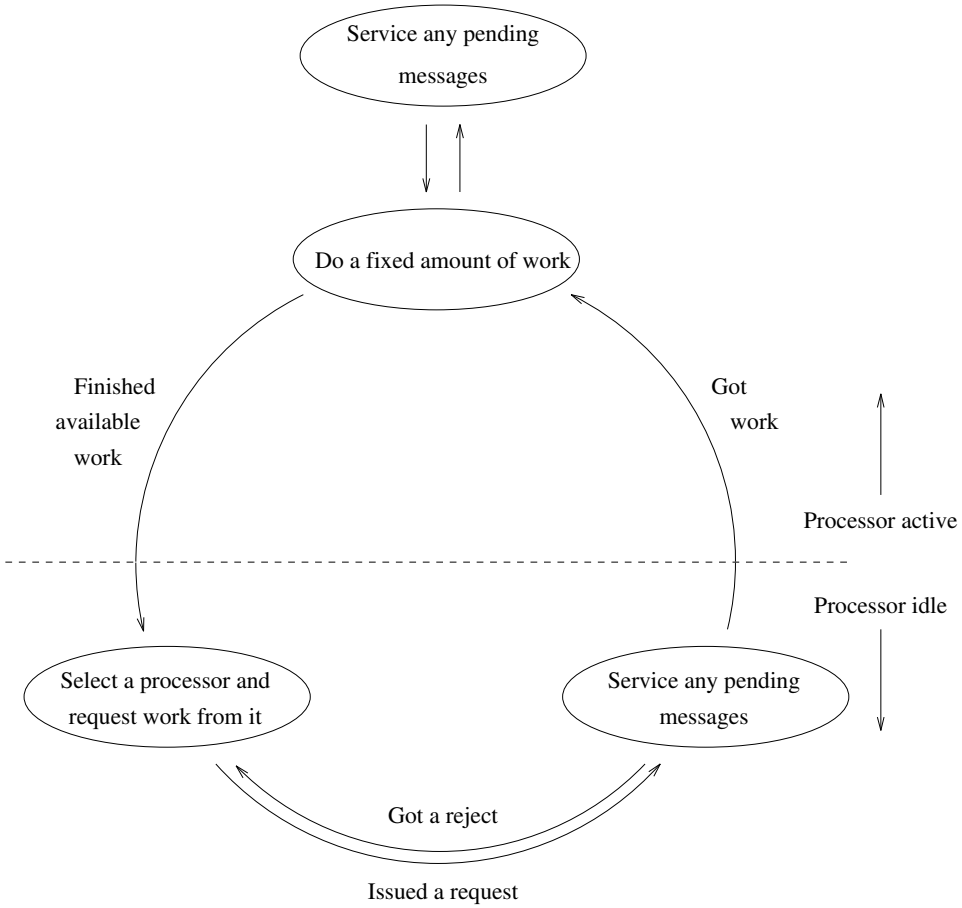
(a) (b)

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

Parallel Depth-First Search: Dynamic Load Balancing

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

Parallel Depth-First Search: Dynamic Load Balancing

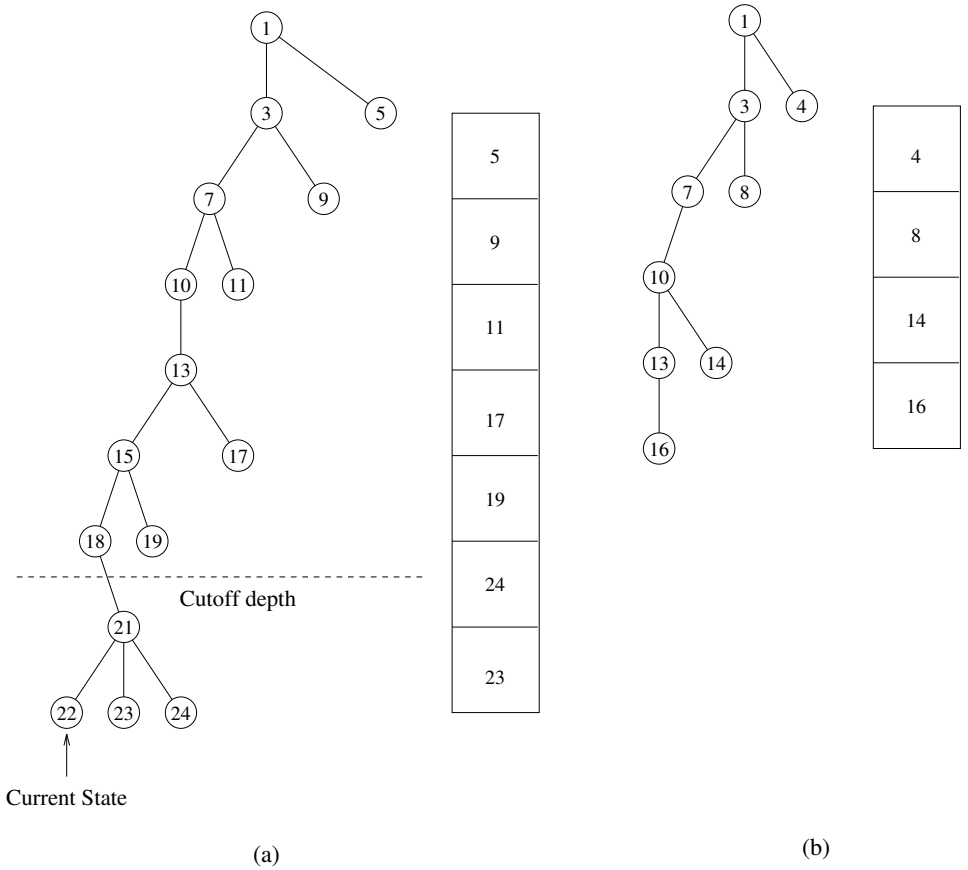


A generic scheme for dynamic load balancing.

Parameters in Parallel DFS: Work Splitting

- Work is split by splitting the stack into two.
- Ideally, we do not want either of the split pieces to be small.
- Select nodes near the bottom of the stack (node splitting), or
- Select some nodes from each level (stack splitting).
- The second strategy generally yields a more even split of the space.

Parameters in Parallel DFS: Work Splitting



Splitting the DFS tree: the two subtrees along with their stack representations are shown in (a) and (b).

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: Each processor maintains a counter and makes requests in a round-robin fashion.
- Global round robin: The system maintains a global counter and requests are made in a round-robin fashion, globally.
- Random polling: Request a randomly selected processor for work.

Analyzing DFS

- We cant compute, analytically, the serial work W or parallel time. Instead, we quantify total overhead T_O in terms of W to compute scalability.

- For dynamic load balancing, idling time is subsumed by communication.

- We must quantify the total number of requests in the system.

Analyzing DFS: Assumptions

- 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 $(1 \quad \psi)w$, there exists an arbitrarily small constant α ($0 < \alpha \leq 0.5$), such that $\psi w > \alpha w$ and $(1 \quad \psi)w > \alpha w$.
- The costant 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 $(1 \quad \alpha)w_i$ 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 $(1 \quad \alpha)W$; after $2V(p)$ requests, it is less than $(1 \quad \alpha)^2 W$.

- After $(\log_{1/(1 \quad \alpha)}(W/\epsilon))V(p)$ 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 given by

$$T_o = t_{comm} V(p) \log W \tag{1}$$

The corresponding efficiency E is given by

$$\begin{aligned} E &= \frac{1}{1 + T_o/W} \\ &= \frac{1}{1 + (t_{comm} V(p) \log W)/W} \end{aligned}$$

Analyzing DFS: $V(P)$ 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.

$V(P)$ for Random Polling

- Let $F(i,p)$ represent a state in which i of the p processors have been requested, and $p \quad i$ have not.

- Let $f(i,p)$ denote the average number of trials needed to change from state $F(i,p)$ to $F(p,p)$ ($V(p) = f(0,p)$).

-

$$
\begin{aligned}
f(i,p) &= \frac{i}{p}(1 + f(i,p)) + \frac{p \quad i}{p}(1 + f(i+1,p)), \\
\frac{p \quad i}{p} f(i,p) &= 1 + \frac{p \quad i}{p} f(i+1,p), \\
f(i,p) &= \frac{p}{p \quad i} + f(i+1,p).
\end{aligned}
$$

$V(P)$ for Random Polling

- We have:

$$\begin{aligned} f(0,p) &= p \times \sum_{i=0}^{p} \frac{1}{p} \frac{1}{i}, \\ &= p \times \sum_{i=1}^{p} \frac{1}{i}, \\ &= p \times H_p, \end{aligned}$$

- As p becomes large, $H_p \simeq 1.69 \ln p$ (where $\ln p$ denotes the natural logarithm of p). Thus, $V(p) = O(p \log p)$.

Analysis of Load-Balancing Schemes

If $t_{comm} = O(1)$, we have,

$$T_o = O(V(p) \log W). \tag{2}$$

- Asynchronous Round Robin: Since $V(p) = O(p^2)$, $T_o = O(p^2 \log W)$. It follows that:

$$\begin{aligned} W &= O(p^2 \log(p^2 \log W)), \\ &= O(p^2 \log p + p^2 \log \log W) \\ &= O(p^2 \log p) \end{aligned}$$

Analysis of Load-Balancing Schemes

- Global Round Robin: Since $V(p) = O(p)$, $T_o = O(p \log W)$. 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:

  $$\frac{W}{p} = O(p \log W) \tag{3}$$

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

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

Analysis of Load-Balancing Schemes

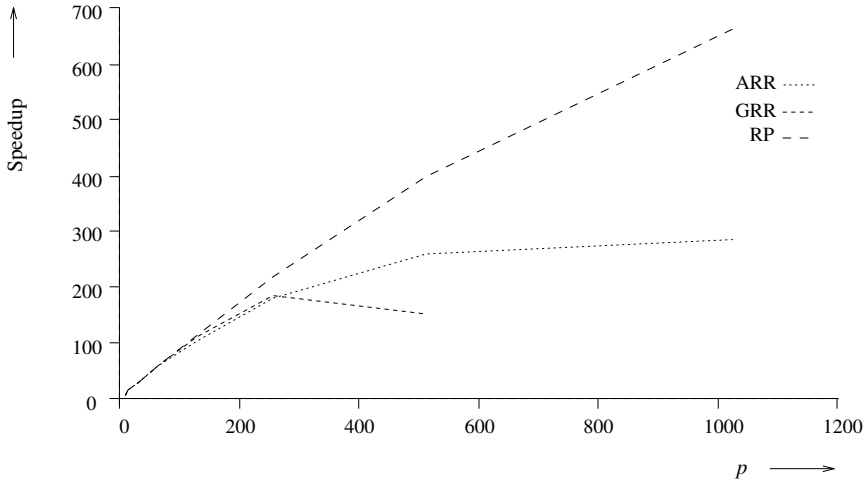
- Random Polling: We have $V(p) = O(p \log p)$, $T_o = O(p \log p \log W)$

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

Analysis of Load-Balancing Schemes: Conclusions

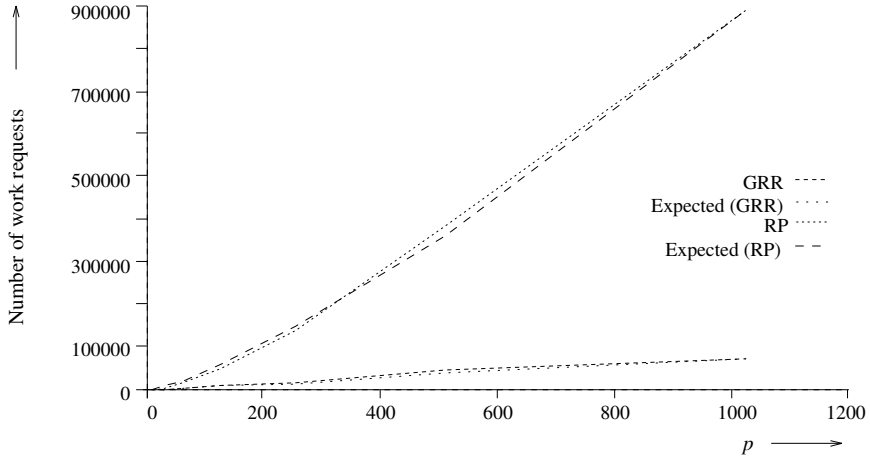
- Asynchronous round robin has poor performance because it makes a large number of work requests.
- Global round robin has poor performance because of contention at counter, although it makes the least number of requests.
- Random polling strikes a desirable compromise.

Experimental Validation: Satisfiability Problem



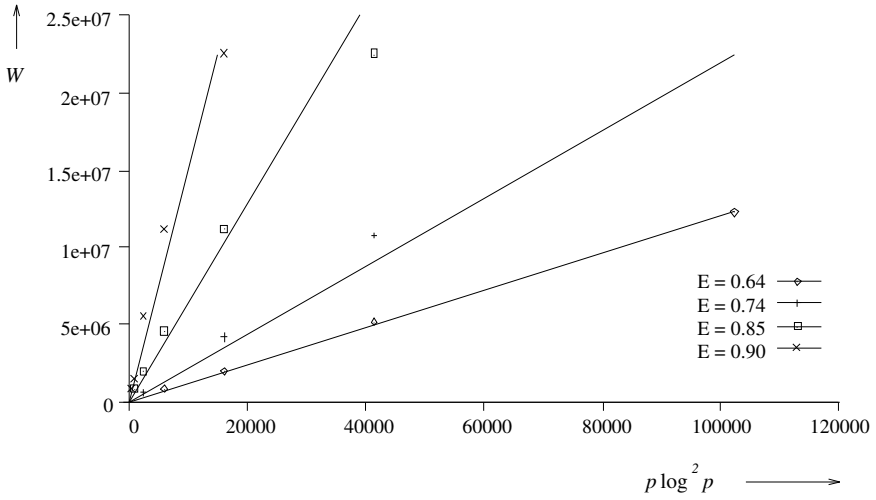
Speedups of parallel DFS using ARR, GRR and RP load-balancing schemes.

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



Experimental isoefficiency curves for RP for different efficiencies.

Termination Detection

- How do you know when everyone's done?
- A number of algorithms have been proposed.

Dijkstra's Token Termination Detection

- Assume that all processors are organized in a logical ring.
- Assume, for now that work transfers can only happen from P_i to P_j if $j > i$.
- Processor P_0 initiates a token on the ring when it goes idle.
- Each intermediate processor receives this token and forwards it when it becomes idle.
- When the token reaches processor P_0, all processors are done.

Dijkstra's Token Termination Detection

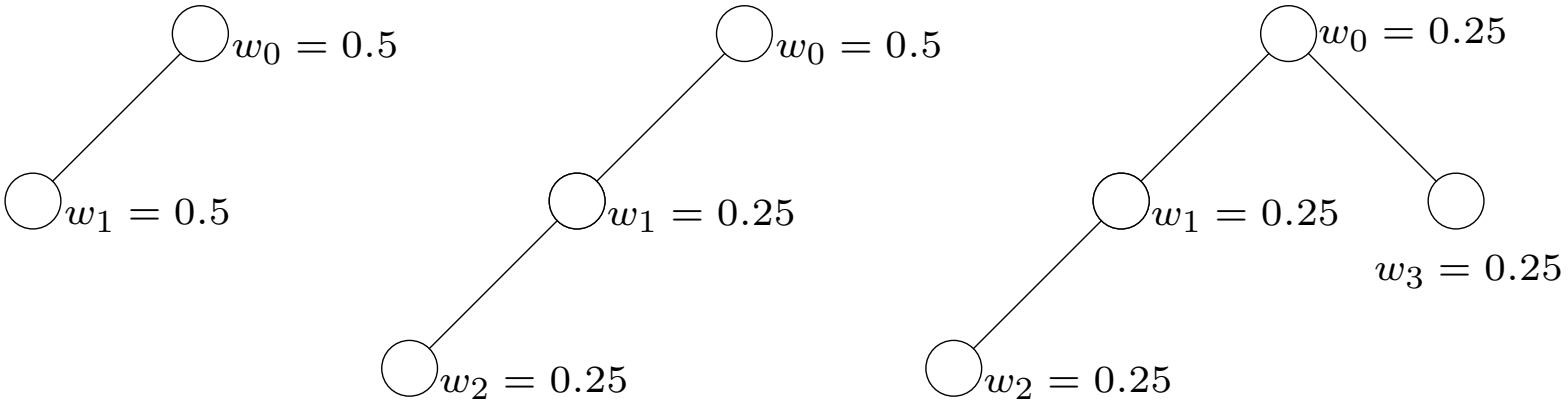
Now, let us do away with the restriction on work transfers.

- When processor P_0 goes idle, it colors itself green and initiates a green token.
- If processor P_j sends work to processor P_i and $j > i$ then processor P_j becomes *red*.
- If processor P_i has the token and P_i is idle, it passes the token to P_{i+1}. If P_i is *red* , then the color of the token is set to *red* before it is sent to P_{i+1}. If P_i is *green*, the token is passed unchanged.
- After P_i passes the token to P_{i+1}, P_i becomes *green* .
- The algorithm terminates when processor P_0 receives a *green* token and is itself idle.

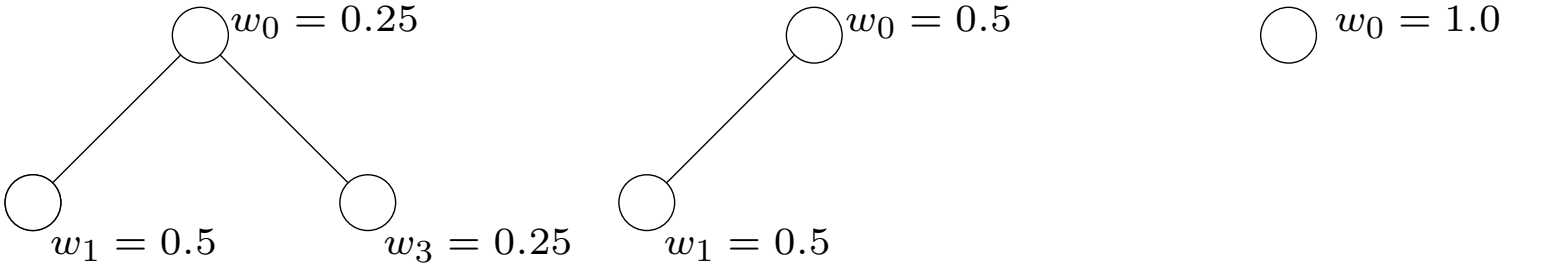
Tree-Based Termination Detection

- Associate weights with individual workpieces. Initially, processor P_0 has all the work and a weight of one.
- Whenever work is partitioned, the weight is split into half and sent with the work.
- When a processor gets done with its work, it sends its parent the weight back.
- Termination is signaled when the weight at processor P_0 becomes 1 again.
- Note that underflow and finite precision are important factors associated with this scheme.

Tree-Based Termination Detection



Step 1 Step 2 Step 3



Step 4 Step 5 Step 6

Tree-based termination detection. Steps 1–6 illustrate the weights at various processors after each work transfer.

Parallel Formulations of Depth-First Branch-and-Bound

- Parallel formulations of depth-first branch-and-bound search (DFBB) are similar to those of DFS.

- Each processor has a copy of the current best solution. This is used as a local bound.

- If a processor detects another solution, it compares the cost with current best solution. If the cost is better, it broadcasts this cost to all processors.

- If a processor's current best solution path is worse than the globally best solution path, only the efficiency of the search is affected, not its correctness.

Parallel Formulations of IDA*

Two formulations are intuitive.

- Common Cost Bound: Each processor is given the same cost bound. Processors use parallel DFS on the tree within the cost bound. The drawback of this scheme is that there might not be enough concurrency.

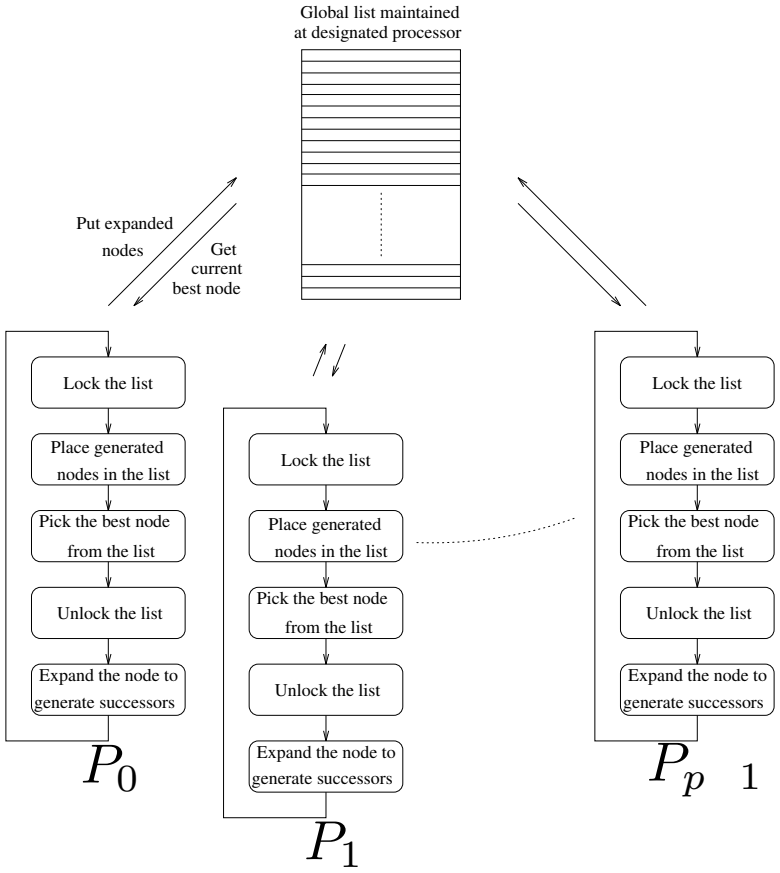
- Variable Cost Bound: Each processor works on a different cost bound. The major drawback here is that a solution is not guaranteed to be optimal until all lower cost bounds have been exhausted.

In each case, parallel DFS is the search kernel.

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



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.

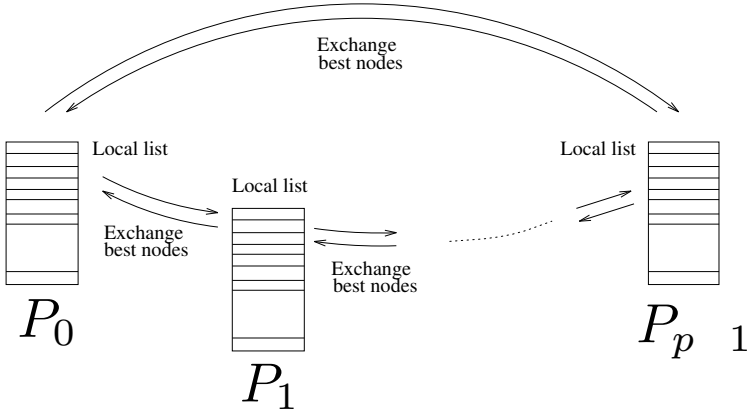
Parallel Best-First Search

- 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

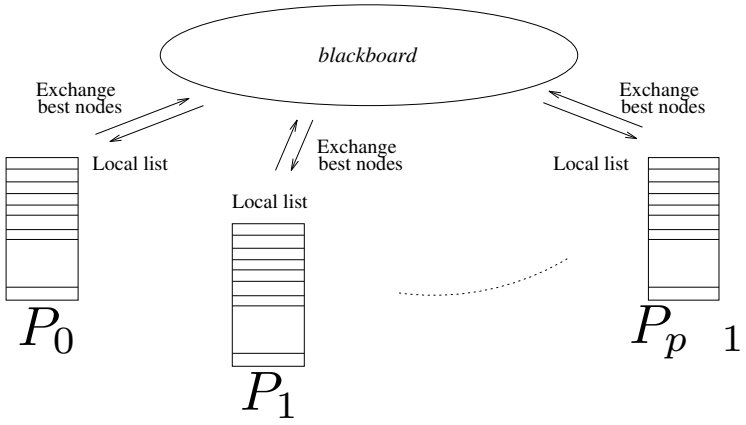
- Avoid contention by having multiple open lists.
- Initially, the search space is statically divided across these open lists.
- Processors concurrently operate on these open lists.
- Since the heuristic values of nodes in these lists may diverge significantly, we must periodically balance the quality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

Parallel Best-First Search



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

Parallel Best-First Search



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

Parallel Best-First Graph Search

- Graph search involves a closed list, where the major operation is a lookup (on a key corresponding to the state).
- The classic data structure is a hash.
- Hashing can be parallelized by using two functions – the first one hashes each node to a processor, and the second one hashes within the processor.
- This strategy can be combined with the idea of multiple open lists.
- If a node does not exist in a closed list, it is inserted into the open list at the target of the first hash function.
- In addition to facilitating lookup, randomization also equalizes quality of nodes in various open lists.

Speedup Anomalies in Parallel Search

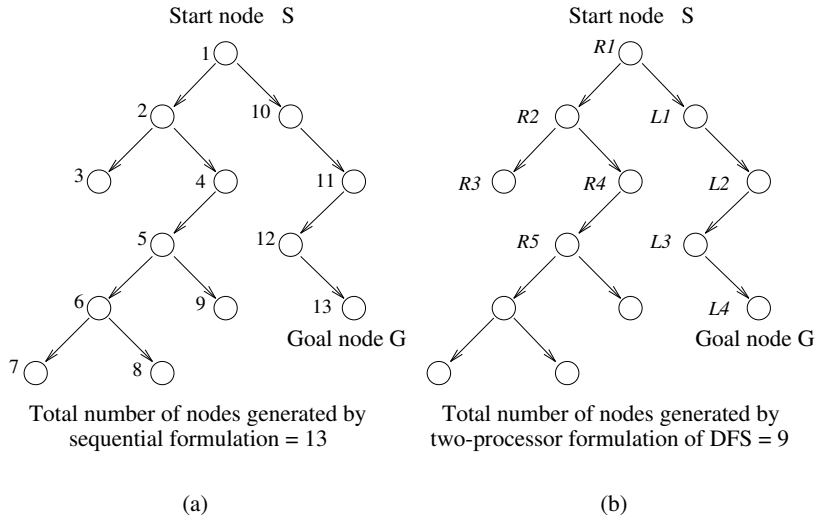
- Since the search space explored by processors is determined dynamically at runtime, the actual work might vary significantly.

- 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.

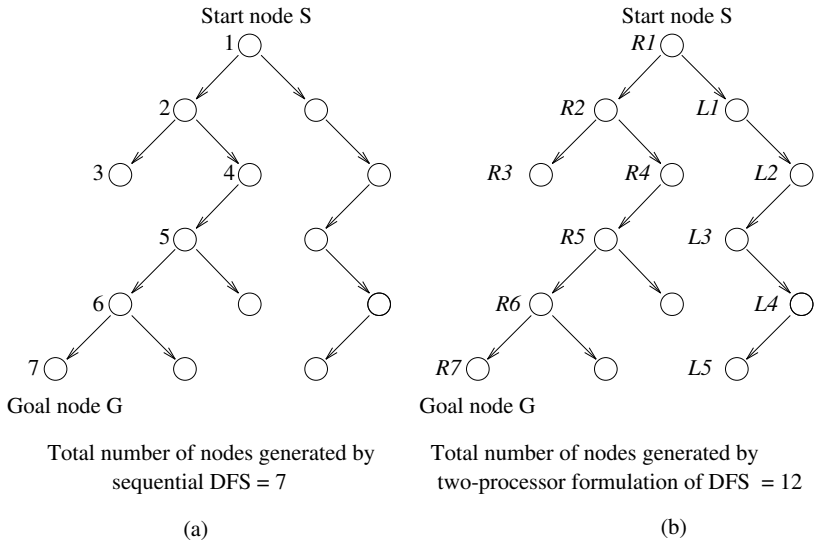
- If the heuristic function is good, the work done in parallel best-first search is typically more than that in its serial counterpart.

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



Total number of nodes generated by sequential DFS = 7

(a)

Total number of nodes generated by two-processor formulation of DFS = 12

(b)

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