Parallel Search Algorithms for Discrete Optimization Problems

Lars Karlsson

2009-05-12

Part I

Introduction

Discrete Optimization Problems

- Given the tuple $(\mathcal{S}, f)$, where
  - $\mathcal{S}$ is a finite set of feasible solutions, and
  - $f : \mathcal{S} \rightarrow \mathbb{R}$ is a cost function,

  the discrete optimization problem (DOP) is to find an optimal solution $s \in \mathcal{S}$ that minimizes f.
- To use search algorithms we need to reformulate the DOP as a problem of finding a shortest path (sometimes *any* path will do) in a graph from a given starting node to one of possibly several goal nodes.
  - Nodes in the graph are called states.
  - The graph is called state space.
  - Goal states represent feasible solutions.
- In contrast to search algorithms, iterative improvement algorithms solve DOPs where the solution is captured by the state itself, rather than the path from a starting node.

Applications

- Puzzle games
  - Towers of Hanoi
  - 15-puzzle
  - Solitaire
- Traveling Salesman Problem
- Integer Programming
- And more...

State Space and Search Tree

State space

- A very important factor is whether the state space is a graph or a tree.
- State space trees are far easier to handle but unfortunately state spaces are often graphs in practice.

Search tree

- The relationship between an expanded state and its successors defines a search tree.
- Note that a state may be expanded several times and hence appear in the search tree several times.
- The size of the search tree is often proportional to th time required by the algorithm.

Why not simply use a Shortest Path Algorithm?

- **Idea:** explicitly form the state space as a graph and use Dijkstra's shortest path algorithm or some other algorithm to find an optimal solution.
- **Problem:** the state space is often enormous and can not be represented explicitly and even if it could it would take too long just to enumerate all the states.
  - The **15-puzzle** has around $16! \approx 10^{13}$ possible configurations. Even using a very compact representation of 64 bits per configuration, the whole state space would occupy roughly 152TB of memory.
- **Solution:** state space represented implicitly using a successor operator which enumerates all successors of a given state.
  - Makes it possible to explore the state space.
  - Enables explicit storage of selected parts of the state space.

Search Overhead in Parallel Search Algorithms

- W = number of states expanded by serial algorithm.
- W_p = number of states expanded by parallel algorithm.
- The search overhead
$$\frac{W_p}{W}$$
describes the overhead due to the order in which states are expanded.
- For uninformed search it is often possible to observe speedup anomalies where
$$\frac{W_p}{W} < 1$$
due to the parallel algorithm searching in multiple regions simultaneously.
- For informed search the situation is reversed and the search overhead is added on top of the usual parallel overheads.

Cost Function and Heuristics

- The cost function is broken down into two components:

$$f(s) = g(s) + h(s).$$

- $f(s)$ is the estimated cost of an optimal solution going through state s.
- $g(s)$ is the cost of reaching state s.
- $h(s)$ is an estimate (heuristic) of the cost of going from state s to the closest goal state.
- If $h(s)$ is an understimate it is said to be an admissible heuristic (which is important for optimality).

Part II

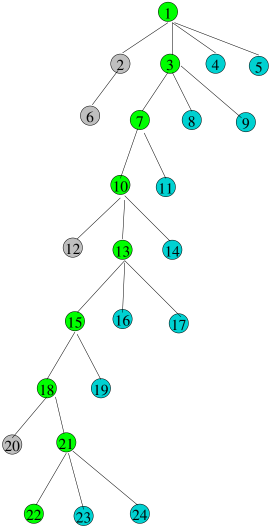
Depth-First Search

Depth-First Search (DFS)

Suitable only for state space trees, since state space graphs are effectively unrolled with a potentially exponential growth of the number of expanded states.

- **Simple backtracking**
  - DFS with termination at first feasible solution.
  - Does not find optimal solution
- **DFS with Branch-and-Bound (DFBB)**
  - Keeps going after finding a solution.
  - Prunes states which can not result in a better solution.
  - Finds optimal solution
- **Iterative Deepening**
  - DFS bounded by depth.
  - Iterated over increasing depths.
  - Does not find optimal solution
- **Iterative Deepening A***
  - DFS bounded by the cost $f = g + h$.
  - Iterated over increasing costs.
  - Finds optimal solution if h is admissible

DFS and Stack Representations



Unexpanded states

5
4
9
8
11
14
17
16
19
24
23

Unexpanded states and their parents

1	4	5
3	8	9
7	11	
10	14	
13	16	17
15	19	
18		
21	23	24

Load Balancing

- Structure of search tree is often irregular.
- Static distribution of work not an option.
- Dynamic load balancing required.

Dynamic Load Balancing

Initiator

- Receiver-initiated.
  - (ARR) Asynchronous Round Robin
    Processors request work in a round robin fashion independently.
  - (GRR) Global Round Robin
    Processors request work in a synchronized round robin fashion.
  - (RP) Random Polling
    Processors request work randomly and asynchronously.
- Sender-initiated (*not discussed here*).

Work splitting

- Send node near bottom of stack. Suitable for uniform search trees since a shallow node is the root of a large subtree.
- Send half of the nodes spread across multiple levels. Suitable for irregular search trees.

Analyzing DFS

- We can compute neither W nor T_p.
- Express T_o in terms of W and use $pT_p = W + T_o$.

Assumptions:

- Communication subsumes idling $\rightarrow$ quantify number of requests.
- Work can be divided into pieces as long as it is larger than a threshold ϵ.
- The work-splitting strategy is reasonable. Whenever work ω is split into two parts $\psi\omega$ and $(1-\psi)\omega$, there exists an arbitrarily small constant $0 < \alpha \leq 0.5$ such that $\psi\omega > \alpha\omega$ and $(1-\psi)\omega > \alpha\omega$. (In effect, the two pieces are not too imbalanced.)

Analyzing DFS

- Consequence of assumptions: if a processor initially had work ω, then after one split neither processor can have more than $(1-\alpha)\omega$ work.
- Let $V(p)$ be the total number of work requests before each process receives at least one work request.
- If the largest piece of work at any time is W, then after $V(p)$ requests, a process can not have more than $(1-\alpha)W$ work (i.e., each process has been the subject of a split at least once).
- After $2V(p)$ requests, no more than $(1-\alpha)^2 W$ work, and so on.
- After $(\log_{1/(1-\alpha)}(W/\epsilon))V(p)$ requests, no processor has more work than the threshold ϵ.
- Conclusion: total number of work requests is

$$\mathcal{O}(V(p)\log W)$$

Analyzing DFS: $V(p)$ for some Load Balancing Schemes

- **Asynchronous Round Robin**: $V(p) = \mathcal{O}(p^2)$.
- **Global Round Robin**: $V(p) = \mathcal{O}(p)$.
- **Random Polling**: Worst case $V(p)$ is unbounded (we analyze average case instead)

Analyzing DFS: $V(p)$ for Random Polling

- Let $F(i,p)$ be a state in which i of the p processes have received a request and $p-i$ have not.
- Let $f(i,p)$ be the average number of trials required to change from state $F(i,p)$ to $F(p,p)$.
- $V(p) = f(0,p)$
- $$\begin{aligned} f(p,p) &= 0, \\ f(i,p) &= \frac{i}{p}(1+f(i,p)) + \frac{p-i}{p}(1+f(i+1,p)), \\ \frac{p-i}{p}f(i,p) &= 1 + \frac{p-i}{p}f(i+1,p), \\ f(i,p) &= \frac{p}{p-i} + f(i+1,p). \end{aligned}$$

Analyzing DFS: $V(p)$ for Random Polling

- Finally, we get

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

- The harmonic series is roughly $1.69 \ln p$, so $V(p) = \mathcal{O}(p \log p)$.

Isoefficiency: ARR

$$T_o = \mathcal{O}(V(p)\log W)$$

Since

$$V(p) = \mathcal{O}(p^2)$$

it follows that

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

Isoefficiency: GRR

$$T_o = \mathcal{O}(V(p) \log W)$$

Since

$$V(p) = \mathcal{O}(p)$$

it follows that

$$W = \mathcal{O}(p \log p)$$

However, this does not account for the contention at the global counter. The counter is incremented $\mathcal{O}(p \log W)$ times in $\mathcal{O}(W/p)$ time.

This gives

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

and

$$W = \mathcal{O}(p^2 \log p)$$

which is the isoefficiency.

Isoefficiency: RP

$$T_o = \mathcal{O}(V(p) \log W)$$

Since

$$V(p) = \mathcal{O}(p \log p)$$

it follows that

$$W = \mathcal{O}(p \log p \log W) = \mathcal{O}(p \log^2 p)$$

Summary of Analysis

- ARR has poor performance due to its many requests.
- GRR suffers from contention.
- RP is a suitable compromise.

Dijkstra's Token Termination Detection

- Processes ordered in logical ring: $P_0, \ldots, P_{p-1}$.
- When P_0 goes idle, it creates a green token and sends it to P_1.
- If process P_i sends work to P_j, $j < i$ (backwards in the ring), then P_i becomes red.
- If process P_i becomes idle and has the token, it sends the token to P_{i+1}. P_i
  - colors the token red if P_i is colored red.
  - leaves the token unchanged if P_i is colored green.
- After P_i sends the token to P_{i+1} it becomes green.
- Termination is detected when P_0 receives a green token.

Overhead of Dijkstra's Token Detection

- Assume detection is initiated when everybody is out of work.
- p steps required.
- $T_o = \Omega(p^2)$
- Isoefficiency:
$$W = KT_o = \Omega(p^2)$$
- Can we do better?

Tree-based Termination Detection

- Processes ordered in logical binary tree.
- P_0, the root, has initially all work and a weight of $w = 1$.
- Each time work is split, the requested process splits its weight in two and sends one half with the response.
- When a process goes idle, it returns its weight to its parent.
- Termination detected when P_0 has $w = 1$ and is idle.
- Numerical difficulties: representing the weight in finite arithmetic requires great care.

Parallel Depth-First Branch-and-Bound

- Very similar to parallel DFS.
- Each process records the best solution found so far which it uses as local bound.
- When a process finds a new best solution it broadcasts it to the other processes.
- Stale local bounds only affect efficiency (i.e., increases the search overhead) and not correctness.

Parallel Iterative Deepening A*

Two intuitive parallel formulations:

- **Common Cost Bound**: all processes use the same cost bound. Parallel DFS used within bound. Might expose too little concurrency.
- **Variable Cost Bound**: to increase the available concurrency, processes work on different cost bounds. When a solution is found it might not be optimal; all lower cost bounds must be examined first. Sequential DFS used by each process.

Part III

Best-First Search

Best-First Search

- The major drawback with DFS is that it does not use heuristics on a global scale and hence searches unintelligently through the state space.
- Another drawback is that DFS is only suitable for state space trees.
- Best-first search overcomes both of these limitations at the expense of using a lot of memory.
- As its name implies, BFS uses heuristics to expand the currently most promising state.
- A^* is a well-known instance of BFS.

Best-First Search

- The central data structure in BFS is the OPEN list (typically a priority queue).
- The OPEN list maintains all known and unexpanded states.
- If the heuristic is admissible, BFS finds an optimal solution.
- For state space graphs, a CLOSED list (typically a hash table) is also required to avoid re-expansion of states.
- If a newly expanded state exists in any of the lists with a better heuristic value, it is not inserted in the OPEN list.

Parallel BFS (centralized list)

Shared-Memory Pseudo-Code (state space tree)

```
while not terminated do
    Lock the OPEN list.
    Place generated nodes in the list.
    Pick the best node from the list.
    Unlock the OPEN list.
    Expand the node to generate successors.
end while
```

Parallel BFS (centralized list)

- Heavy contention on the `OPEN` list.
- Let t_{access} be the time spent accessing the list.
- Let t_{expand} be the time spent expanding a node.
- Sequential runtime: $n(t_{\text{access}} + t_{\text{expand}})$ for n nodes.
- Parallel runtime at least nt_{access} due to contention.
- Speedup bounded above by

$$S_p \leq \frac{t_{\text{access}} + t_{\text{expand}}}{t_{\text{access}}}$$

- Example: $t_{\text{expand}} = 9t_{\text{access}} \Rightarrow S_p \leq 10$.

Parallel BFS: Distributed `OPEN` Lists

- Contention reduced by having multiple `OPEN` lists.
  - k processes share one list.
  - extreme case: one list per process ($k = 1$).
- The quality of the nodes in the lists may diverge and thus some processes may spend considerable time expanding unpromising states.
- Quality equalization strategy required to avoid quality divergence.
  - Random
  - Ring-based
  - Blackboard
  - And more...

Parallel BFS: State Space Graphs

- For state space graphs, the `CLOSED` list is required, and it need to be distributed to avoid contention.
- Two-level hashing of states potentially solves the three problems of
  - a distributed `CLOSED` list,
  - quality equalization, and
  - load balancing.
- Two-level hashing:
  - States hashed to processes with 1st has function
  - States hashed in `CLOSED` list with 2nd has function
  - Expanded nodes are sent to owner (1st hash function)
  - Owner process checks against its `CLOSED` list and inserts the state into its own `OPEN` list if necessary.

Summary

- Search requires dynamic load balancing.
- Random polling is more scalable than asynchronous round robin (too many requests) and global round robin (contention).
- Termination detection necessary. Dijkstra's token detection scheme. Tree-based detection scheme.
- Depth-first search requires little memory but searches inefficiently and has a high computational overhead on state space graphs.
- Best-first search requires much memory but focuses first on promising states and can handle state space graphs.
- Two-level hashing in BFS solves the contention, load balancing, and quality equalization problems.

Part IV

Applications

Application: 15-puzzle

- Each state is a configuration.
- A piece move transitions between states.
- The state space is a graph with a large depth, so plain DFS is not appropriate.
- Heuristics available (e.g., sum of Manhattan distances), so BFS or IDA* would be appropriate.

Application: 0/1 Integer Programming

- Linear programming problem: minimize

$$f(x) = c^T x$$

  where the variables $x_i \in \{0, 1\}$.
- The variables are subject to linear constraints:

$$Ax \geq b$$

- Formulation as search problem:
  - Assign values to x_i for $i = 1, \ldots, n$.
  - Goal states at depth n (all variables assigned) with all constraints satisfied.
  - State space is a tree, so DFS applicable.

Application: 0/1 Integer Programming

- If the subset x_a of the variables have been assigned values and x_u have not, we can partition A such that

$$\hat{A}x_u \geq b - \tilde{A}x_a$$

- For each constraint

$$\hat{a}_i^T x_u \geq b_i - \tilde{a}_i^T x_a,$$

  the LHS is bounded above by the sum of all positive elements in $\hat{a}_i$.
- If any such upper bound fails to satisfy its constraint, then the state can never result in a feasible solution and a subtree of the search tree can be pruned.
- DFBB can be used in conjunction with this bound.