



# Parallel Search Algorithms

CS121 Parallel Computing  
Fall 2024

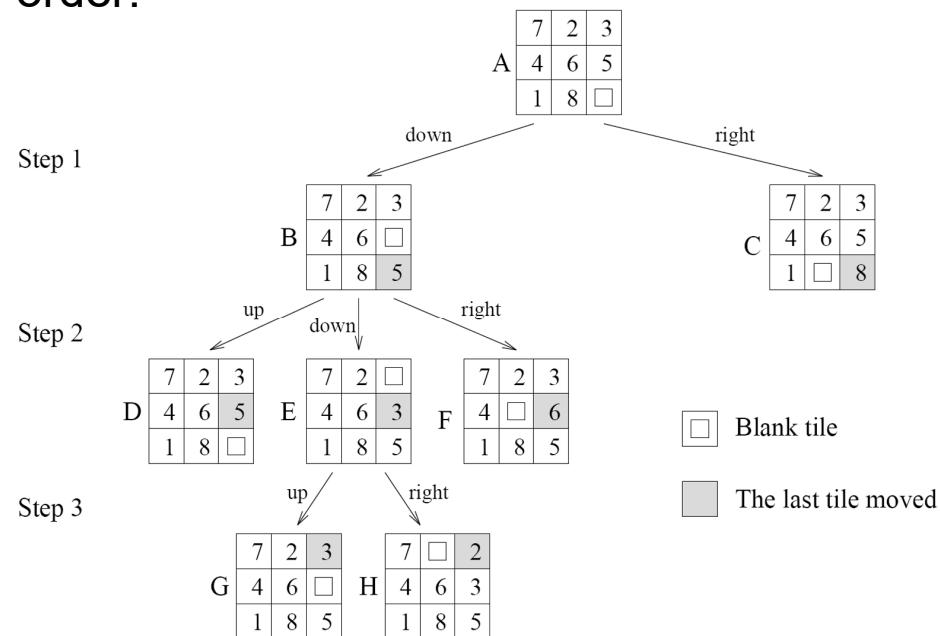


# Discrete optimization problems

- A discrete optimization problem (DOP) consists of a tuple  $(S, f)$ , where  $S$  is a (finite or infinite) set of feasible solutions satisfying certain constraints, and  $f$  is a cost function for each solution,  $f: S \rightarrow \mathbb{R}$ .
- **Ex** Planning and scheduling, optimal layout in VLSI, logistics and control, satisfiability problems.
- We consider the minimization problem, i.e. find  $x^* \in S$  s.t.  $\forall x \in S: f(x^*) \leq f(x)$ .

# Searching for solutions

- Many interesting DOPs are NP-hard, so we either rely on heuristics or search algorithms.
- Searching a DOP can be modeled by a graph, with nodes (aka states) being (possibly infeasible) solutions and edges between states that are “close enough”.
  - Cost of states can be modeled using weights on nodes and edges.
  - Search process traverses states in some order to find min cost feasible state.
    - State with no successor in traversal called terminal state. Otherwise it's a nonterminal state.
- **Ex** 8-puzzle. From starting configuration, move blank tile to lower right so all tiles are in order.



# Searching for solutions

- **Ex** A mixed integer program (MIP) consists of a linear objective function and a system of linear constraints, where some of the variables are required to be integers.

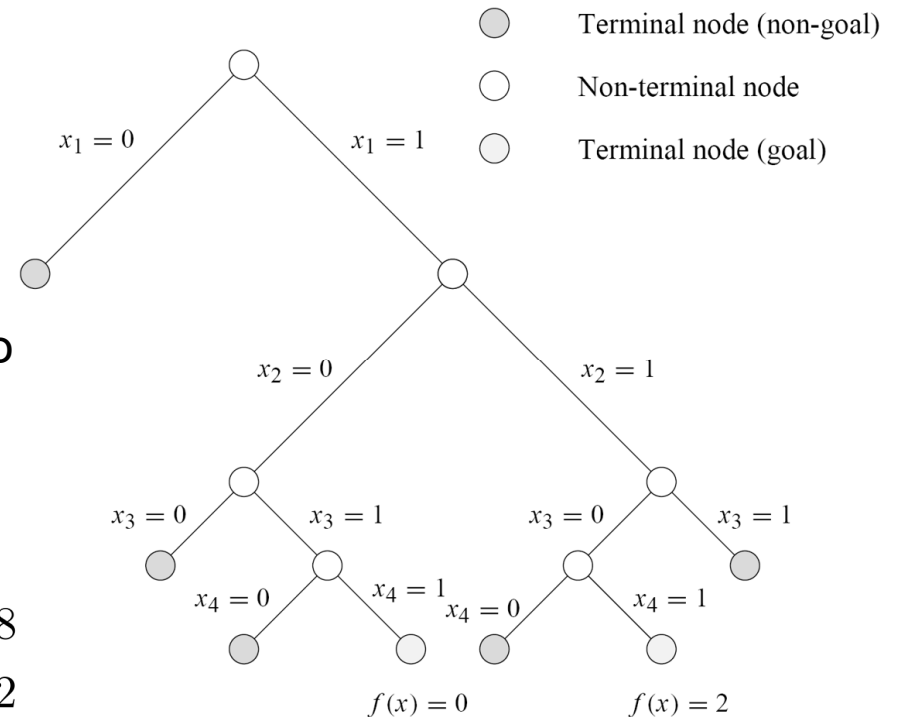
- States are assignments of values to the variables.
- MIPs can model many interesting DOPs, including NP-hard ones.

□ **Ex**  $\min 2x_1 + x_2 - x_3 - 2x_4 \quad \text{s.t.}$

$$\begin{aligned} 5x_1 + 2x_2 + x_3 + 2x_4 &\geq 8 \\ x_1 - x_2 - x_3 + 2x_4 &\geq 2 \\ 3x_1 + x_2 + x_3 + 3x_4 &\geq 5 \end{aligned}$$

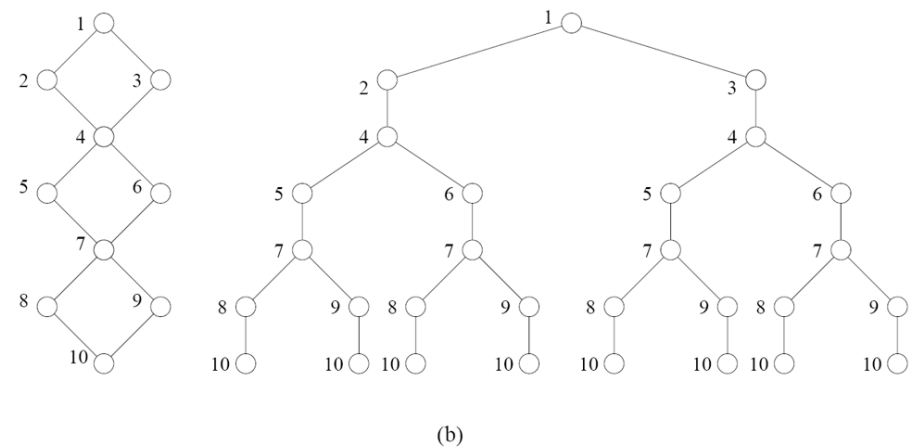
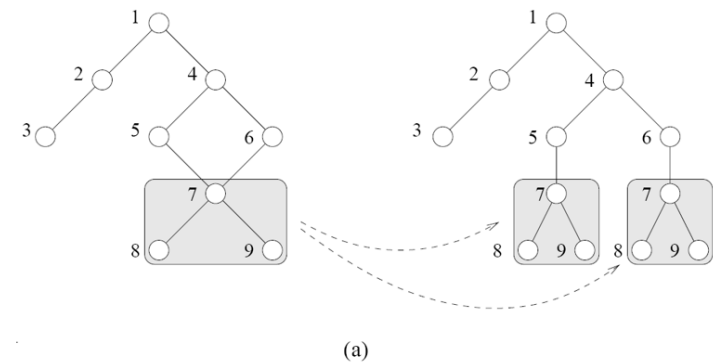
- Feasible solutions must satisfy

$$\sum_{x_j \text{ is free}} \max\{A[i, j], 0\} + \sum_{x_j \text{ is fixed}} A[i, j]x_j \geq b_i, i = 1, \dots, m$$



# Structure of search graph

- When the search graph is a tree, there's only one way to arrive at each state, and the state has a unique cost determined by the root-state path.
- When the search graph is not a tree, there can be multiple paths to each state.
- Can unfold graph into a tree to use tree based search methods.
  - Sometimes unfolded tree much larger than original graph.
- Same state may be discovered multiple times.
  - Use duplicate detection to detect if node has already been explored.
    - Ex Store explored nodes in hash table.
  - Cost to the state is the minimum among all the paths.
    - Update the cost as different paths discovered.
    - Ex Hash table store both a state and its current min cost.



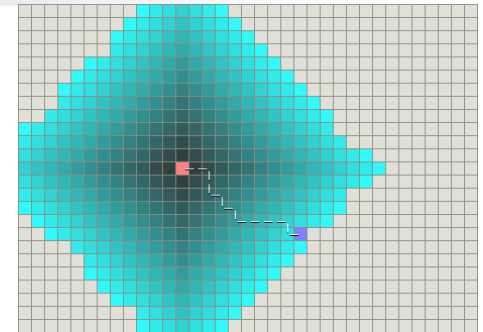


# Branch and bound

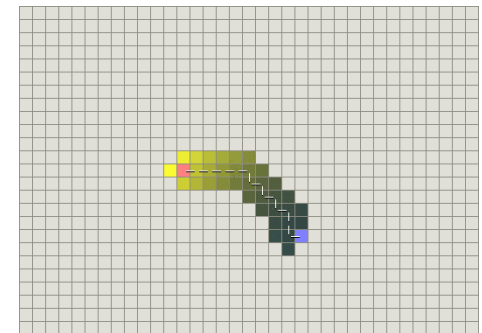
- Traverse the search graph in some order. At any time, we have a frontier of deepest nodes that have been explored.
- For each node  $p$  on the frontier, maintain a pair  $(m(p), M(p))$ , where  $m(p) \leq \min$  possible value of any feasible descendant of  $p$ , and  $M(p) \geq \max$  possible value of any feasible descendant of  $p$ .
  - $m$  and  $M$  can be computed using a fast inexact heuristic.
  - **Ex** In knapsack,  $m$  can be the value of current knapsack,  $M$  can be the value if all remaining items are placed in knapsack.
- If  $m(p) \geq M(q)$  for nodes  $p, q$  on the frontier, then don't need to explore any descendants of  $p$ .
  - The best (smallest) solution starting from  $p$  is worse (larger) than the worst possible solution starting from  $q$ , so we shouldn't explore  $p$ .

# A\* search

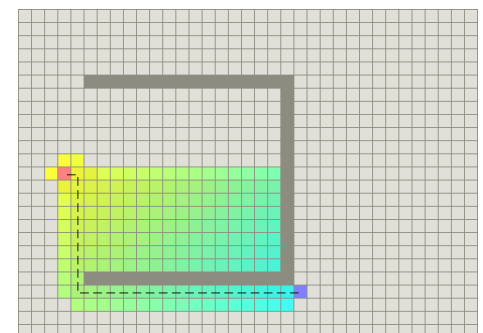
- An improvement of Dijkstra's shortest path (i.e. min cost) algorithm that searches "most promising" paths first.
- For each node  $n$  in search graph, let
  - $g(n)$  = cost from root to  $n$ .
  - $h(n)$  = a lower bound on the cost from  $n$  to a solution (aka admissible heuristic).
  - $f(n) = g(n) + h(n)$ , i.e. a lower bound on a solution from  $n$ .
- A\* search expands nodes in order of nondecreasing  $f(n)$ .
  - Dijkstra's expands nodes in nondecreasing order of  $g(n)$ .
- Can be implemented in a similar way to Dijkstra.
  - Keep a closed list of explored nodes. Their values are already minimal.
  - Keep an open list of nodes whose values are tentative and can still decrease.
  - Repeatedly explore min  $f(n)$  node. Then update  $g$ ,  $h$  and  $f$  functions.
- Guaranteed to find min cost solution.
- If  $h(n)$  close to real cost from  $n$  to a solution, then A\* searches few nodes and is fast.
  - Ex  $h(n)$  for grid search can be Manhattan distance from  $n$  to goal.



Dijkstra's algorithm



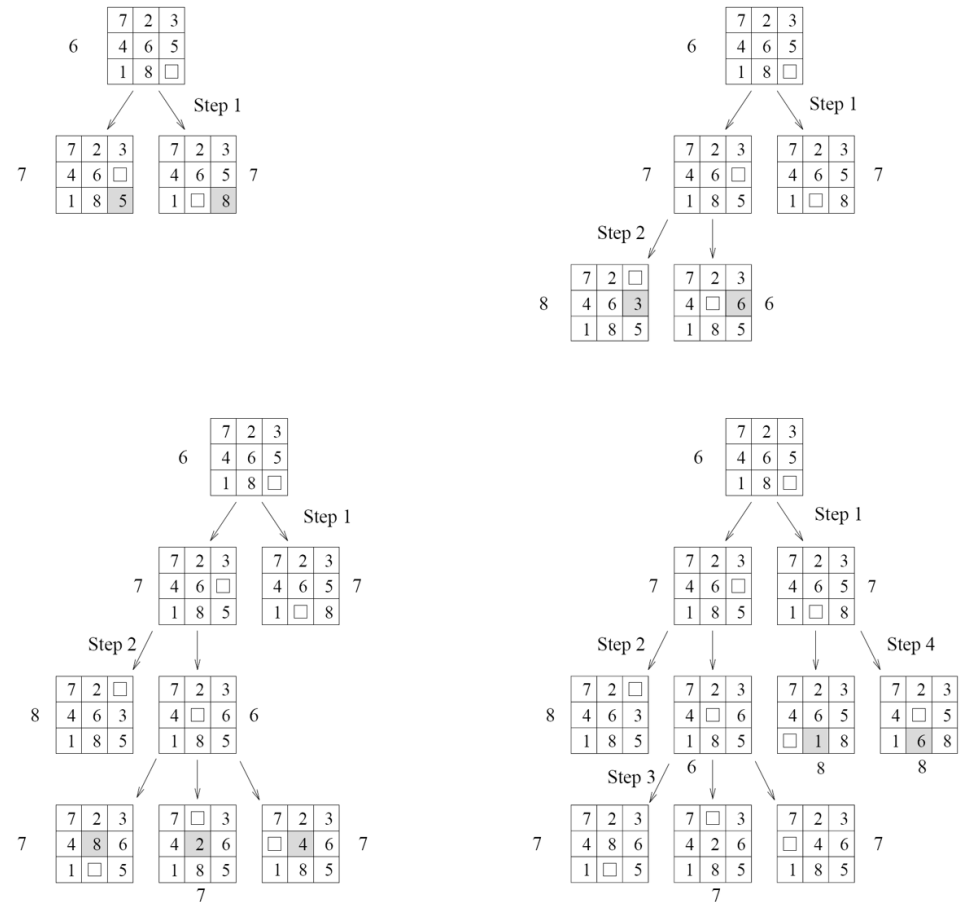
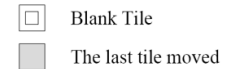
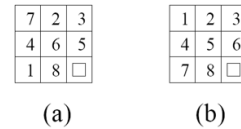
A\* search with Manhattan distance heuristic



A\* search with an obstacle

# A\* search

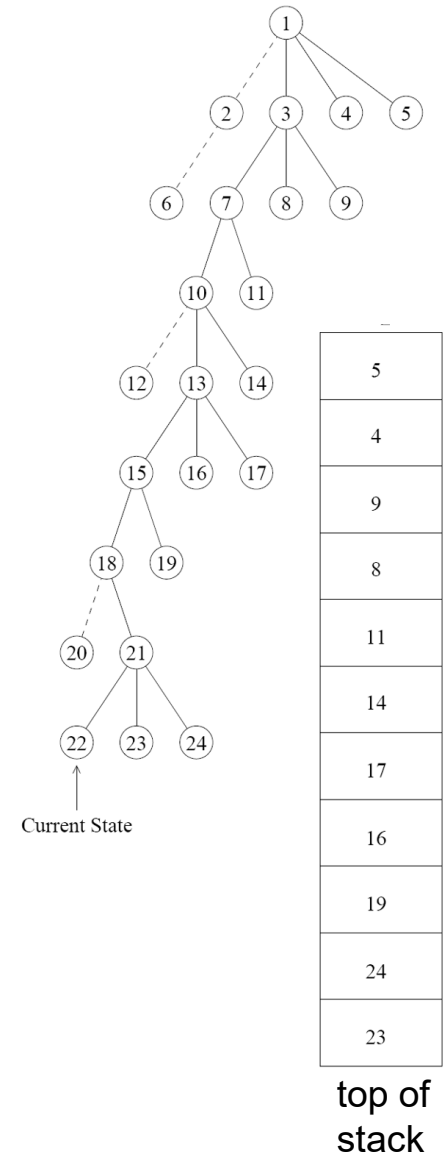
- **Ex** For 8 puzzle, one possible  $h_1(n)$  is number of tiles in incorrect positions.
- **Ex** Another  $h_2(n)$  is the sum of the Manhattan distances of each tile from its final position.
  - Each move of the blank tile moves one numbered tile one position.
- Decreases number of states searched by many orders of magnitude.
- Can also use  $h(n)$  that's slightly larger than cost from  $n$  to goal.
  - Further reduces number of nodes searched.
  - Can produce somewhat suboptimal solution.





# DFS and variants

- Depth first search
  - Do DFS on the search graph. Backtrack from infeasible terminal nodes.
  - Visit node at top of stack. Add new unvisited nodes to top of stack.
- Depth first branch and bound
  - Search in DFS order, but cut off branches using BB.
- Iterative deepening A\*
  - DFS can search very deep in part of the tree, whereas a better solution exists higher up.
  - Instead, do multiple rounds of DFS, each round searching deeper.
  - For each search, set an upper bound B for cost. If current node n has  $f(n) > B$ , then backtrack.
    - On the next round, increase B to smallest non-explored  $f(n)$  value from last round.
    - For first round, set  $B = f(\text{root})$ .
- Other more advanced algorithms used in practice include cutting plane methods and branch and cut.



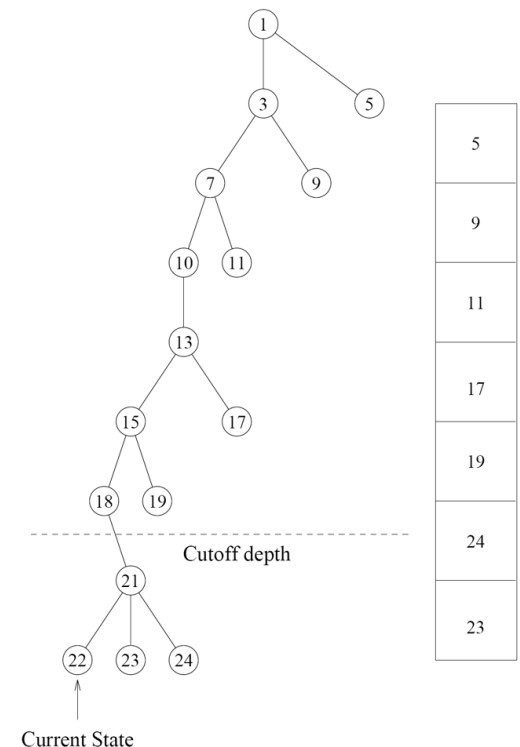
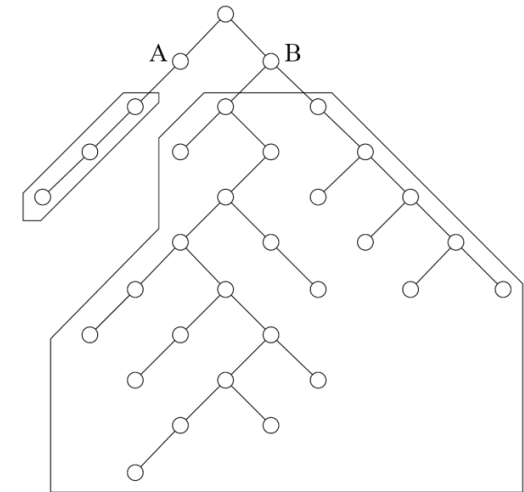


# Parallel search

- Tree and graph searches can be very slow, so we can parallelize these algorithms.
- Parallel search algorithms have several sources of overhead, including communication, load imbalance and contention for shared data structures.
- Parallel algorithms can explore different part of search tree than the sequential algorithm, since the exploration order is different.
- Let  $W_s, W_p$  be the number of nodes explored by sequential and parallel algorithms, resp.
  - Search overhead factor is  $\frac{W_p}{W_s}$ .
  - Overhead may be  $>$ ,  $=$  or  $<$  than 1.
    - As discussed later, when overhead  $< 1$ , we have a speedup anomaly.

# Parallel DFS

- To parallelize DFS, can assign different processors parts of the DFS stack to explore.
- Static assignments cause poor load balancing, since different branches have different sizes.
- Hard to estimate the sizes of branches. So instead, do dynamic load balancing.
  - Can use e.g. the work-stealing algorithm from previous lecture.
- Control how much to steal.
  - Stealing too little causes overhead from many steals.
  - Stealing too much can cause poor load balancing.
- Also choose who to steal from.
  - In asynchronous round robin, processors choose victims independently, and round robin through them.
  - In global round robin, processors maintain victim in a global variable, and round robin through them.
  - In randomized work stealing, nodes pick victim randomly.
- To avoid sending very small amounts of work, set a cutoff depth and don't send work past cutoff.

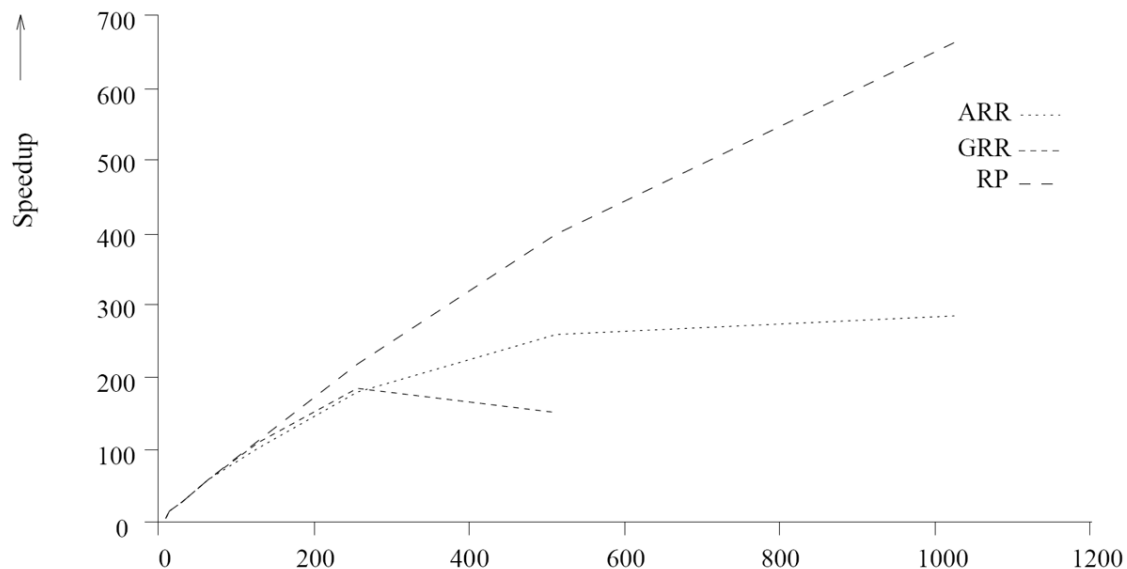


# Load balancing overhead

- Since load is generated dynamically, it's hard to estimate the precise overhead. So we make the following simplifying assumptions.
  - If victim processor has  $W$  amount of work, after work stealing victim and stealing processors each have  $\geq \alpha W$  amount of work, for  $0 < \alpha \leq \frac{1}{2}$ .
    - I.e. the work stealing roughly balances the work between the thief and victim.
    - After load balancing, both processors have  $\leq (1 - \alpha)W$  work.
  - If a processor has  $\leq \epsilon$  work, for a small  $\epsilon > 0$ , it doesn't do load balancing.
- Suppose there are  $p$  processors, and let  $V(p)$  be total number of steals before every processor receives one steal attempt.
  - $V(p)$  depends on the particular work stealing algorithm.
- Suppose the max amount of work at any processor is currently  $W$ . Then after  $V(p)$  steals, the max amount is  $\leq (1 - \alpha)W$ , by assumption above.
  - After  $2V(p)$  steals, max work at any processor is  $\leq (1 - \alpha)^2 W$ . In general, after  $kV(p)$  steals, it's  $\leq (1 - \alpha)^k W$ .
  - So after  $\log_{\frac{1}{1-\alpha}} \left( \frac{W}{\epsilon} \right) V(p)$  steals, each processor has  $\leq \epsilon$  amount of work.
  - So total overhead from work stealing is  $O(V(p) \log W)$ .
  - Efficiency is  $1 / (1 + \frac{t_{comm} V(p) \log W}{W})$ .

# Load balancing overhead

- For asynchronous round robin,  $p \leq V(p) \leq p^2$ , because each processor round robins through victim processors.
  - For isoefficiency need  $W = \Omega(V(p) \log W) = \Omega(p^2 \log p)$ .
- For global round robin,  $V(p) = p$ .
  - So the victim variable is accessed  $O(p \log W)$  times.
  - There is contention on shared victim variable. Total variable access time is  $O(p \log W)$ .
  - For isoefficiency, we want each process's execution time  $\geq$  variable access time.
    - So  $\frac{W}{p} = \Omega(p \log W)$ , and  $W = \Omega(p^2 \log p)$ .
- For randomized, can prove expected  $V(p) = O(p \log p)$ .
  - For isoefficiency need  $W = \Omega(p \log^2 p)$ .





# Parallel DFBB and IDA\*

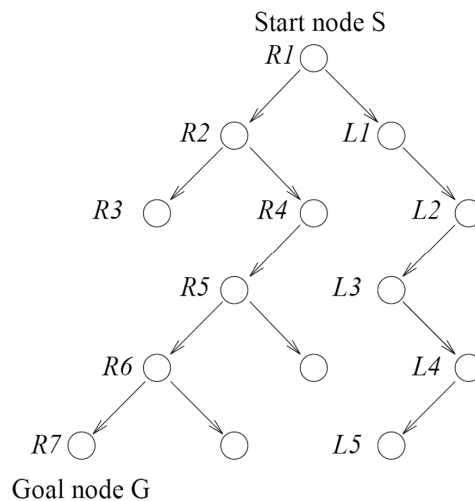
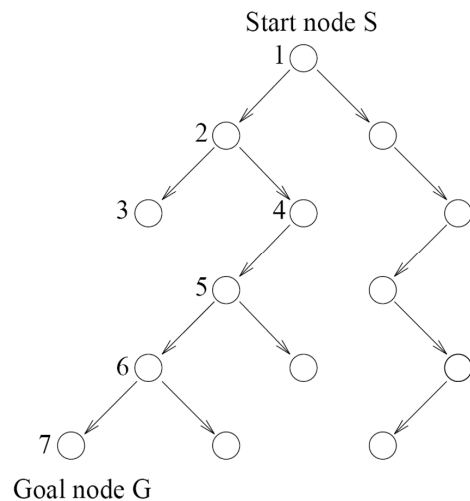
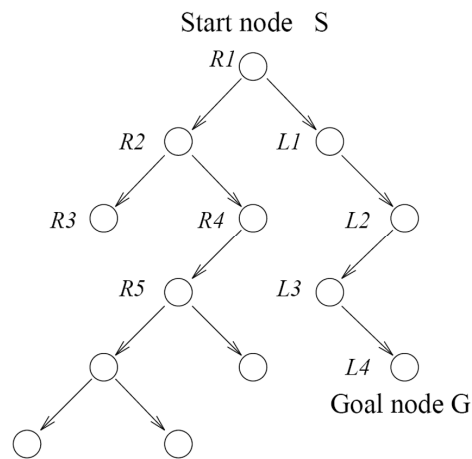
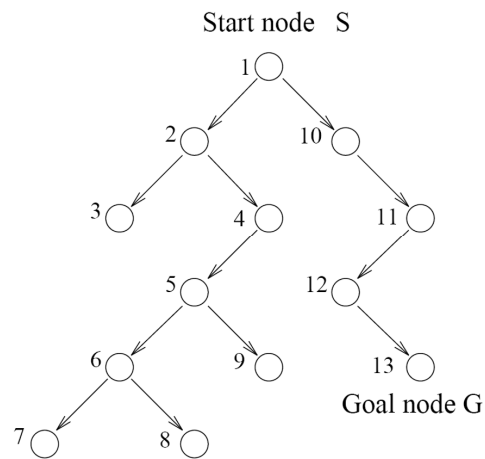
- To parallelize depth first branch and bound, processors store the globally best solution.
  - If a processor finds a better solution, it broadcasts it.
  - Since the best solution changes infrequently, there's not much overhead.
- For parallel IDA\*, broadcast the bound for each round of DFS, then parallelize the DFS.



# Parallel A\*

- In A\*, the cost of nodes are stored in a priority queue.
  - Parallelizing A\* requires accessing the queue in parallel.
- With  $p$  processors, expand smallest  $p$  nodes in queue.
  - This expands some nodes that aren't expanded in the sequential algorithm, leading to redundant work.
- Another problem is contention when accessing the queue.
  - To decrease contention, can give each processor its own copy of the queue.
  - If a processor finds a good node (with small  $f(n)$ ), it should update other nodes so they don't expand non-minimal nodes.
  - More updates reduces redundancy but increases communication, so need to find a tradeoff.
- If the search graph is not a tree, then can visit the same node multiple times, so need duplicate detection.
  - Can store visited nodes in a hash table and distribute hash table among processors.
  - Causes communication for each node visited.
    - Can be amortized if amount of computation per node is large.

# Speedup anomalies



- Since parallel search explores nodes in a different order than sequential search, it can explore fewer or more nodes.
- Ex In the upper figures, the sequential search on the left explores 13 nodes before finding the goal. On the right, parallel search using two processors R and L explores 9 nodes.
  - There is a superlinear speedup of  $13/5 > 2$ .
  - This is called an acceleration anomaly.
- Ex In the lower figures, the left sequential search explores 7 nodes, but the right parallel search explores 12 nodes.
  - This is called a deceleration anomaly.





# Average speedup in DFS

- In certain simple settings we can analyze the expected speedup using parallel DFS.
- Assume the following
  - The search graph is a tree with  $M$  leaves. All the solutions are at the leaves, and there is at least one solution.
  - The number of nodes explored is proportional to the number of leaves explored.
  - The parallel DFS partitions the tree into  $m$  (= number processor) equal parts. All processors run at the same speed.
  - The sequential DFS orders the partitions randomly and searches each completely before searching the next partition.
  - In the  $i$ 'th partition, each leaf has an independent probability  $\rho_i$  of being a solution.
  - Both sequential and parallel DFS stop after finding one solution.

# Average speedup in DFS

- The expected number of leaves (and hence nodes) searched in the  $i$ 'th partition is  $1/\rho_i$ .
- The expected running time  $W_s$  of sequential DFS is proportional to  $\frac{1}{m} \left( \frac{1}{\rho_1} + \frac{1}{\rho_2} + \dots + \frac{1}{\rho_m} \right)$ , since it searches a random partition.
- In parallel DFS, in each parallel step one leaf from each partition is searched. The probability that at least one of them is a solution (and hence DFS stops) is  $1 - \prod_{i=1}^m (1 - \rho_i) \approx \rho_1 + \rho_2 + \dots + \rho_m$  for small  $\rho$ 's.
  - The expected running time  $W_m$  of parallel DFS is proportional to  $\frac{1}{\rho_1 + \rho_2 + \dots + \rho_m}$ .
- $W_m = O(1/(\text{arithmetic mean of } \rho_1, \dots, \rho_m))$ , and  $W_s = O(1/(\text{harmonic mean of } \rho_1, \dots, \rho_m))$ .
  - Since arithmetic mean  $\geq$  harmonic mean, then  $W_m \leq W_s$ .
  - $W_m = W_s$  only if  $\rho_1 = \rho_2 = \dots = \rho_m$ .
  - If the  $\rho$ 's aren't equal, then  $W_m < W_s$ . So we can get superlinear speedup if the solutions aren't equally distributed in the search partitions.