UNIT 1: State Space Search and Scheduling
Course Goals

Scheduling is a critical activity in many industries:

- **Specialty Manufacturing**. Many small job shops supply companies such as Boeing, HP, GM, etc. They receive orders for the manufacture of individual components or subsystems which they satisfy using a variety of NC milling machines, injection-molding machines, etc.
- **Industrial Assembly**. In assembly tasks where each unit is subject to a large number of customer choices, proper scheduling is critical to high productivity.
- **NASA Shuttle Program**. Each shuttle mission involves thousands of activities that must be performed prior to launch and after landing. Several shuttle missions are in process simultaneously.
- **Airline Gate Scheduling**. Assignment of individual flights to gates is a real-time scheduling task. A flight with many missed connections can cost many thousands of dollars.

An Abstract View of Scheduling

- **A** *scheduling problem* **is a set of jobs to be scheduled using a fixed set of resources.**

- **A** *job* **is a set of tasks and a deadline.** The job may also have a **ready time** and a penalty for tardiness.

- **A** *task* **is a single activity.** It has a **duration**, a set of **resource requirements**, and a list of **prerequisite tasks**. In some scheduling situations, the task may also have some **state requirements**. There may be alternative ways of completing the activity, each with its own set of requirements.

- **A** *resource* **is a re-usable object with a fixed capacity.** Examples of resources are milling machines, fork lifts, airline gates, warehouses, technicians, engineers.

- **A** *schedule* **assigns a start time to each task.** If there are alternative ways of completing a task, the schedule includes exactly one of them.

- **A** *feasible schedule* **is a schedule in which no deadline is missed, no resource is over-committed, and every task starts after its prerequisite tasks have completed.**

Optimal Schedules

In some applications, any feasible schedule is acceptable.

In other applications, the goal is to find the shortest feasible schedule.

In applications where there are penalties for missing deadlines, the goal is typically to find a schedule that minimizes the total penalty.

Finally, in cases where tasks may have alternative ways of being performed (each with its own cost), the goal is to find the cheapest feasible schedule.

An Example Scheduling Problem: Space Shuttle Payload Processing

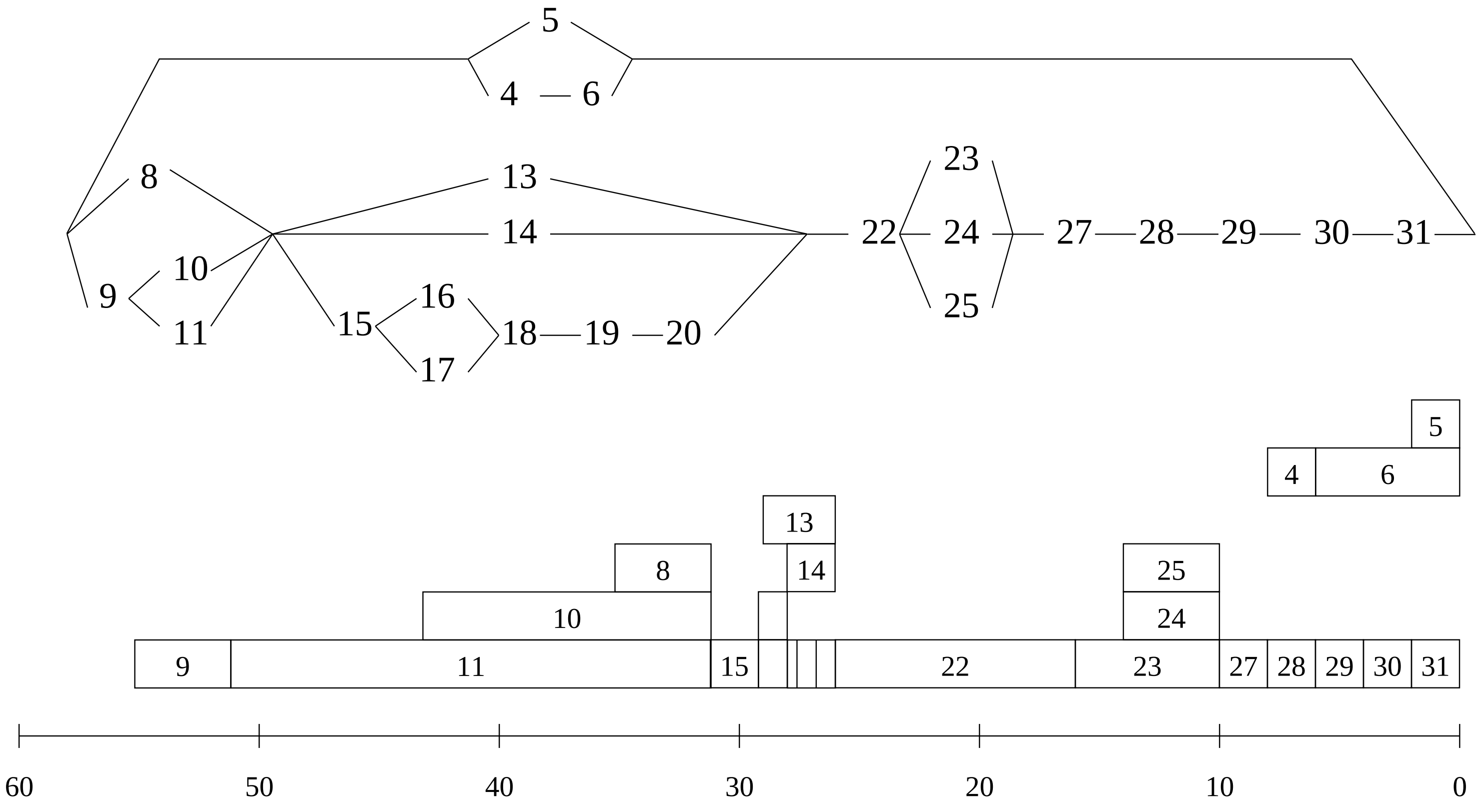
A **job** is a shuttle mission.

Examples of tasks are

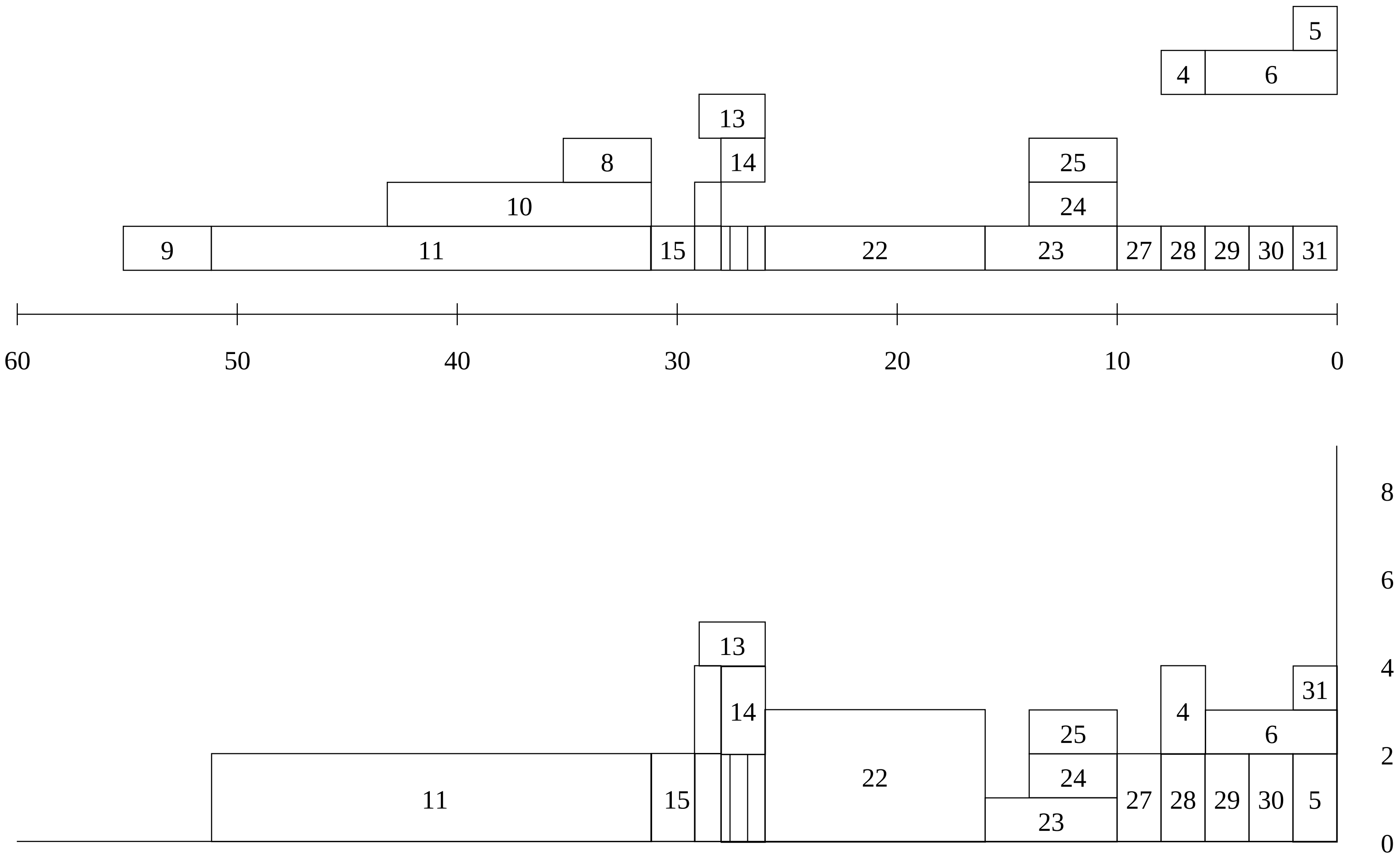
```
Task 16: INSTALL-400-HZ-S-S-INVERTER
  duration 1.2 hours
  requires
    2 technicians
    1 quality-control-officer
  prerequisite tasks: 15

Task 22: EXPERIMENT-INSTALLATION
  duration 10 hours
  requires
    3 technicians
    2 quality-control-officers
  prerequisite tasks: 13, 14, 20
```

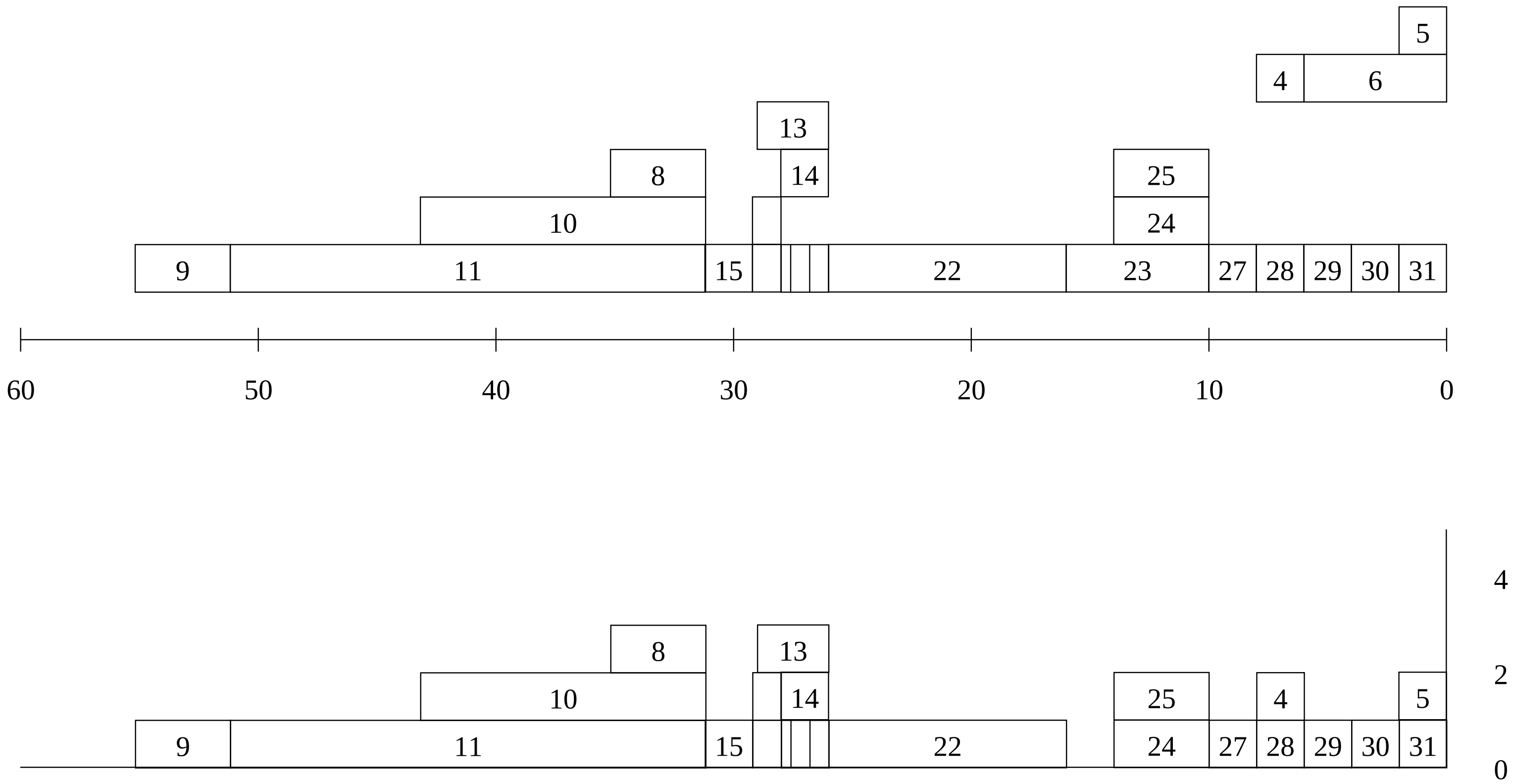
Portion of One Shuttle Mission



Number of Technicians Required



Number of Quality Control Officers Required



NASA Scheduling Goal

A typical shuttle payload scheduling problem involves 3–4 simultaneous shuttle missions where each mission has 140–180 tasks.

Each mission has its own deadline (its launch date).

The goal is to find the shortest feasible schedule. (i.e., the schedule that starts as late as possible and still makes the launch date).

Problem Space Search

A *problem space* **consists of**

- **A** *set of states*
- **A** *designated starting state*
- **A** *set of operators*
- **A** *set of goal states*

Example: Chess

state = a board position

starting state = the starting board position

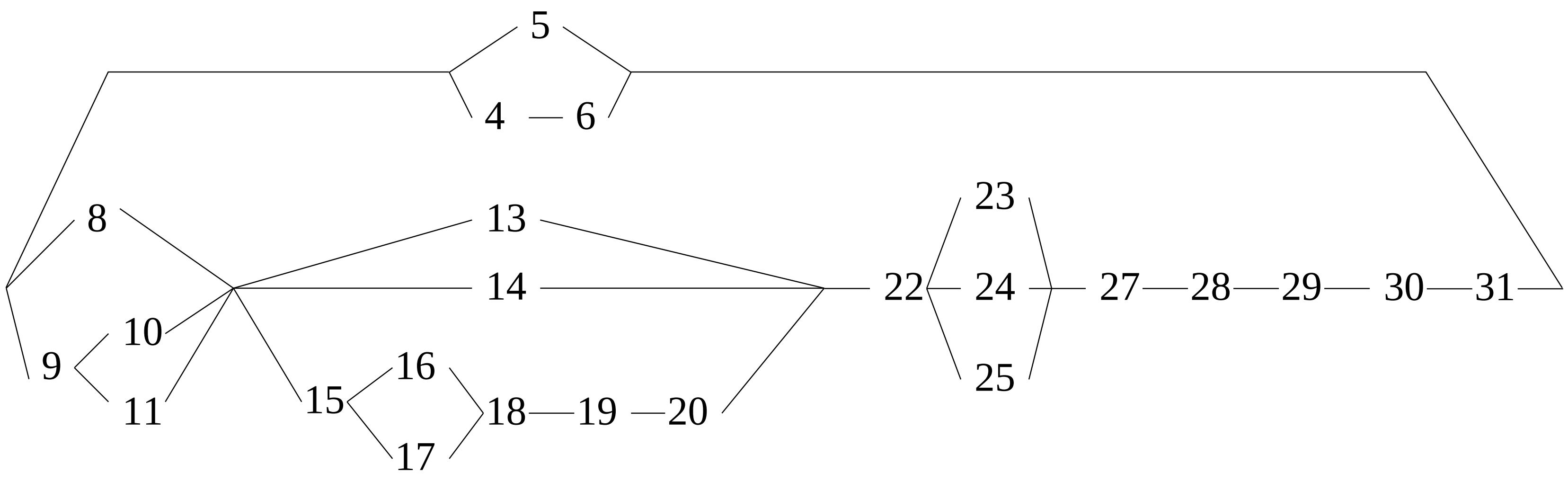
operators = legal chess moves

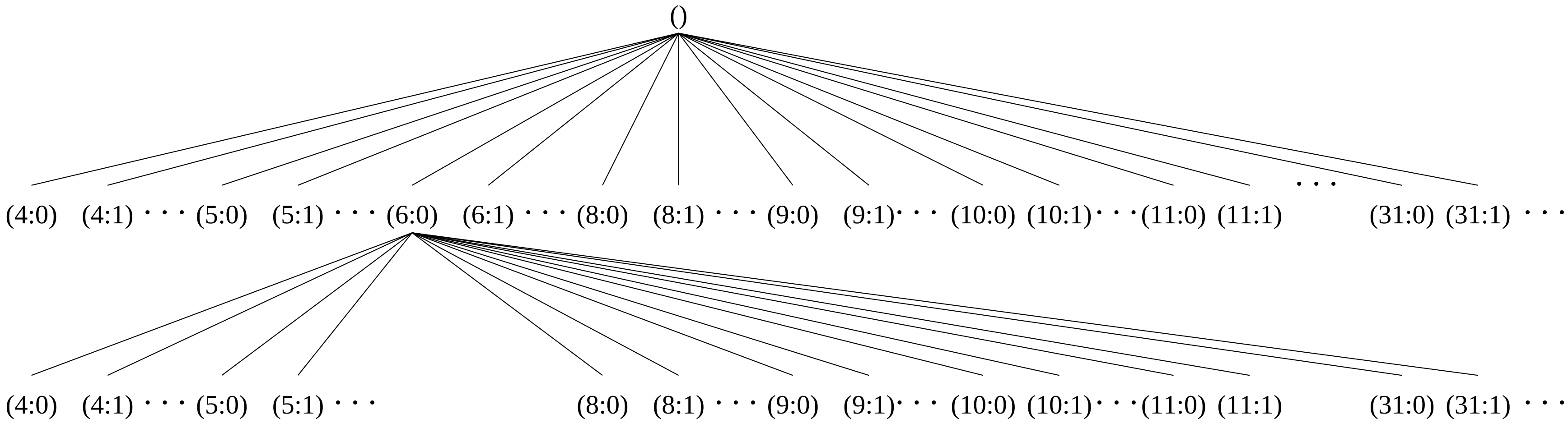
goal state = checkmate

A Problem Space for Scheduling

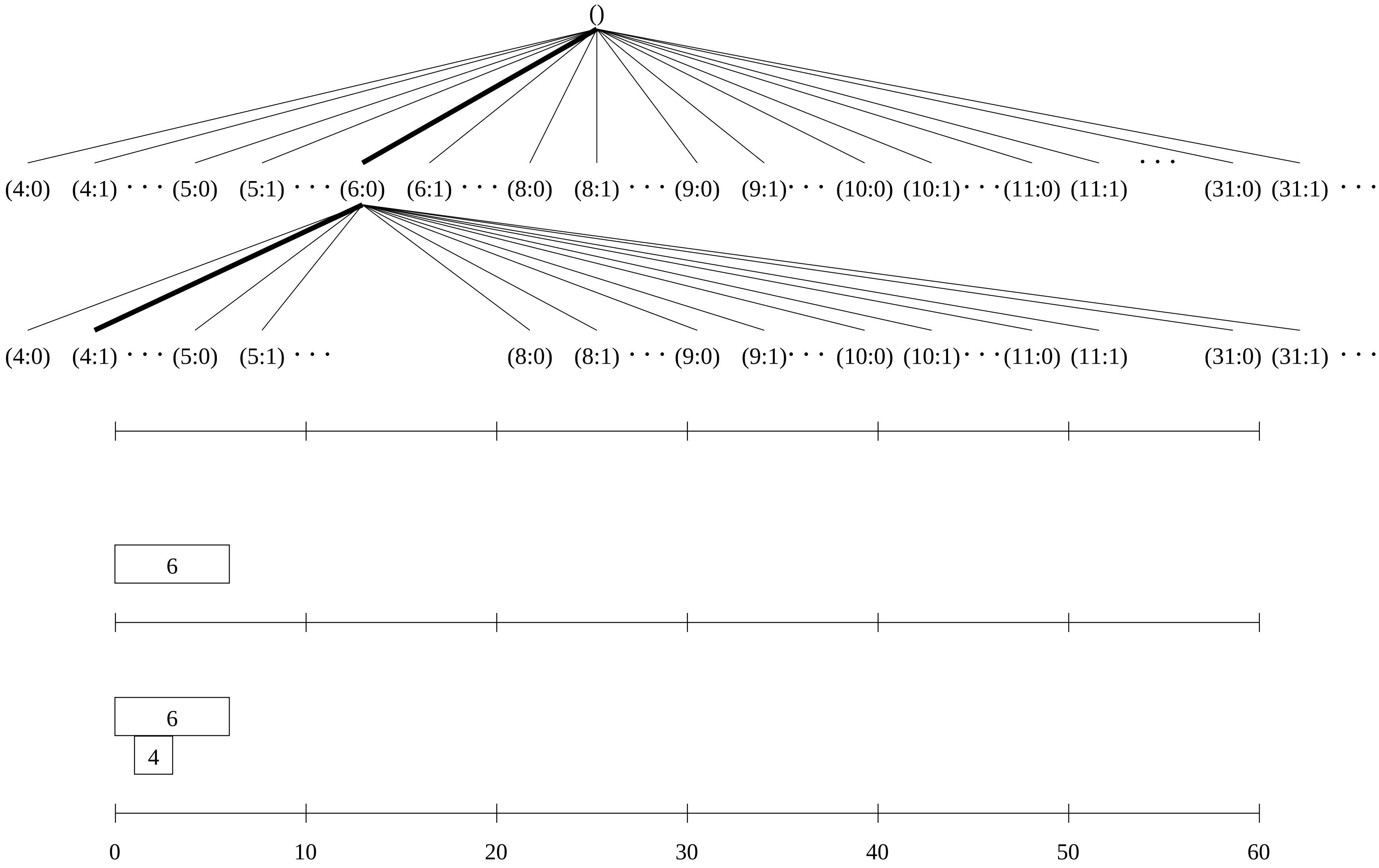
- **state:** a partial schedule in which some of the tasks may have been assigned start times.
- **start state:** the empty schedule (no tasks have start times).
- **operators:** choose a task that is not yet scheduled and choose a start time for it.
- **goal:** the shortest complete schedule such that (a) each task starts after all of its prerequisites have finished and (b) no resources are overallocated.

Searching the Problem Space





One Path in the Problem Space



Properties of the Problem Space Graph

- **Branching Factor**. How many choices are there at each state?
  In our case, the branching factor is infinite, because each task can be scheduled at infinitely many times.
- **Solution Depth**. How many operators must be applied to reach a feasible solution?
  In our case, the solution depth = number of tasks.

Total search space size is B^D, where B is the branching factor and D is the solution depth.

More Properties of the Problem Space Graph

- **Solution Density**. What fraction of the leaf nodes are feasible solutions?

  In our case, most leaf nodes will have constraint violations.

- **Redundancy**. Are there multiple paths to reaching the same state?

  Yes. In our space, if a state has n tasks scheduled, there are $n!$ possible paths to reach that state.

- **Completeness**. Is every possible feasible solution generated?

  Yes.

Improving the Problem Space

- **Remove redundancy**. Order the tasks. Process the tasks according to this ordering. (Note that this does not affect the start times chosen for the tasks, only the order in which they are processed by the problem space.)
- **Incorporate feasibility constraints**. Never schedule a task at a time that would violate a prerequisite constraint or a resource constraint. Process the prerequisites of a task before processing the task itself.
- **Bound the schedule length**. Let TD be the total duration of all tasks. No schedule should ever be longer than this. Hence, we should never schedule a task such that it would end after TD.

Topological Sorted Order

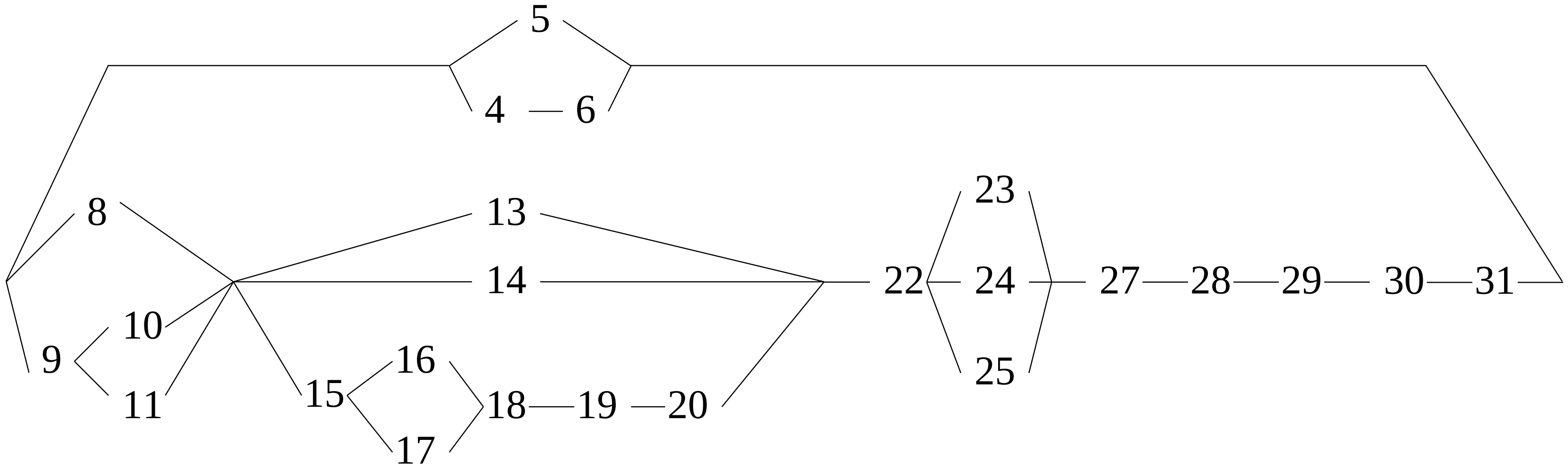
List the tasks of the prerequisite graph in an order such that all prerequisites of a task come before the task itself. In our case, the numerical ordering is a topological order:

4, 5, 6, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 27, 28, 29, 30, 31.

Here is another topological ordering:

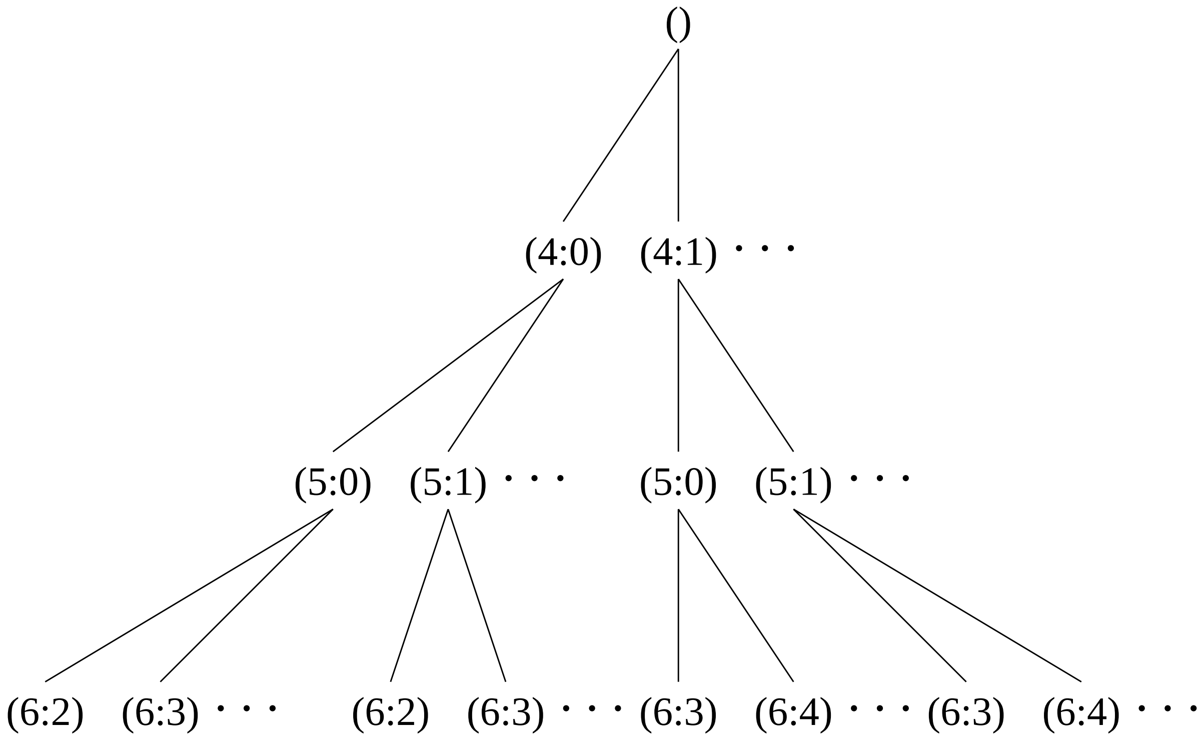
9, 8, 11, 10, 15, 14, 13, 17, 16, 18, 19, 20, 5, 4, 6, 22, 25, 24, 23, 27, 28, 29, 30, 31.

The topological ordering is easy to compute from the prerequisite structure.



New operator description

- **Choose the first unscheduled task according to the topological order.**
- **Choose a start time $\geq$ finish time of the last prerequisite and $\leq TD-$ duration of the task.**



Problem Space Graph Properties

- **Branching Factor**. If N is the number of tasks and TD is the number of possible start times, then the branching factor is certainly always less than $N \times TD$. Because of the topological ordering constraint, the NASA problem has only 1 eligible task at each node. However, $TD = 238$, so the branching factor is always less than 238. The average branching factor is approximately 25. It starts around 140 but rapidly decreases to 1 by depth 12.
- **Solution Depth**. This is 24. So the state space has no more than $238^{24} \leq 1.1 \times 10^{57}$ states. (Probably more like 10^{18}).
- **Solution Density**. Every complete schedule is a feasible solution.
- **Redundancy**. None.

Further Reducing the Branching Factor

Suppose the best feasible schedule found so far has length L. Suppose that in an as-late-as-possible critical path schedule, task T_9 starts 20 time steps before the end of the schedule. We will call this quantity the **critical path margin** for task T_9.

Then in any schedule better than the current best, T_9 cannot start at a time greater than $L - 20$.

This can further constrain the set of possible start times to consider for a task.

Search Algorithms

There are many different search algorithms that have been invented and studied in artificial intelligence. All algorithms can be viewed as dividing the states into three sets:

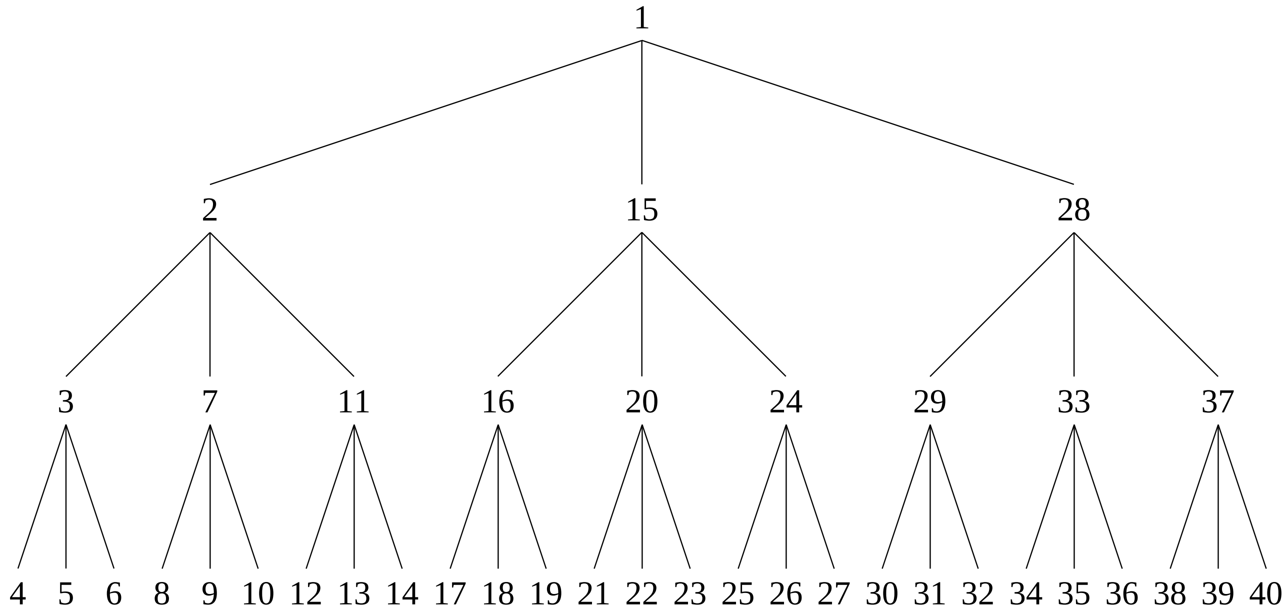
- **Already Visited**. States that have already been visited by the search.
- **Frontier**. States that have been constructed but not yet visited.
- **Unknown.** States that have not yet been constructed.

Search algorithms differ primarily by the way in which they choose states from the frontier and expand them.

Example: Depth-First Search

```
frontier.add(startstate);
do {
  // remove node most recently added to frontier
  current = frontier.pop();
  if (current is feasible solution) {
    if (solution length is better than current best solution) best = current;
    }
  else {
    // expand current by putting all of its children onto the frontier
    for each child node of current {
      frontier.add(child);
    }
} while (! frontier.isEmpty());
```

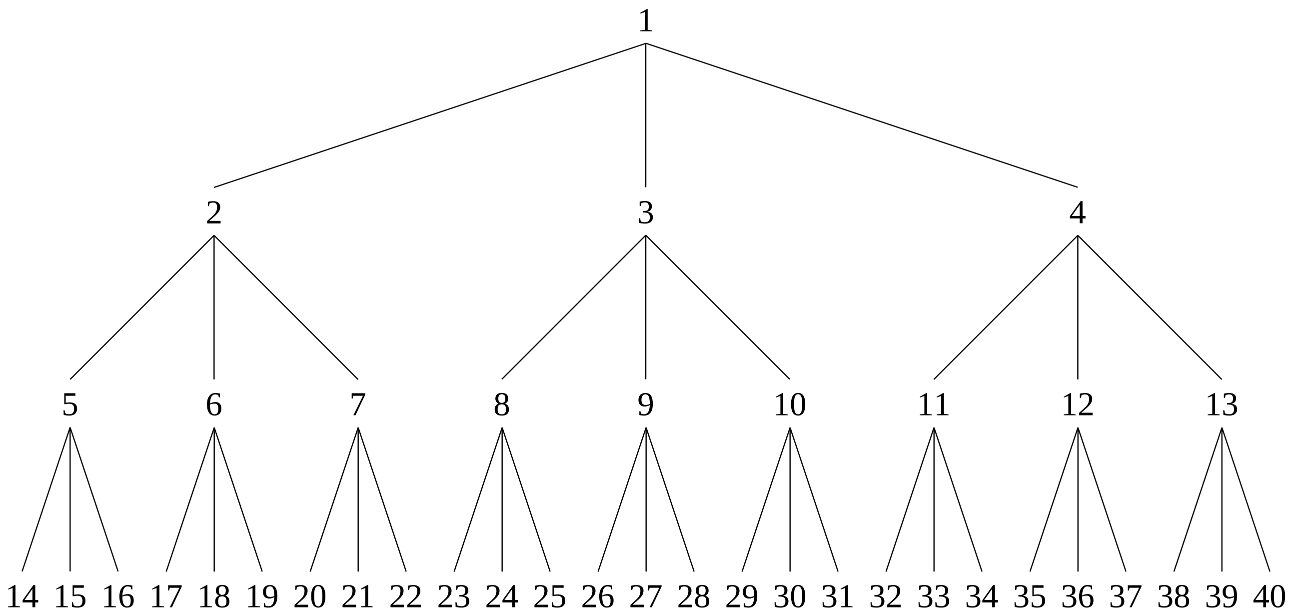
Depth-First Search Graph



Best when solution density is high, search depth is deep, and there are no infinite descending paths.

Breadth-First Search

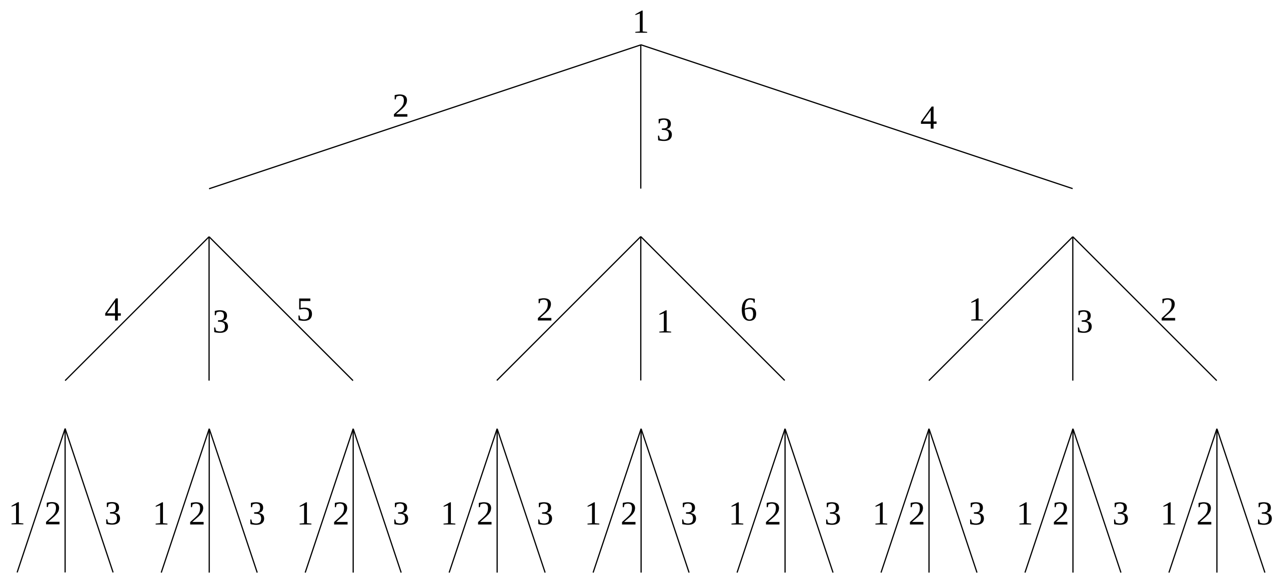
Identical to DFS except that at each point we remove from the frontier the **oldest** node on the frontier.



Hardly ever used because it requires too much memory to store the frontier.

Best-First Search

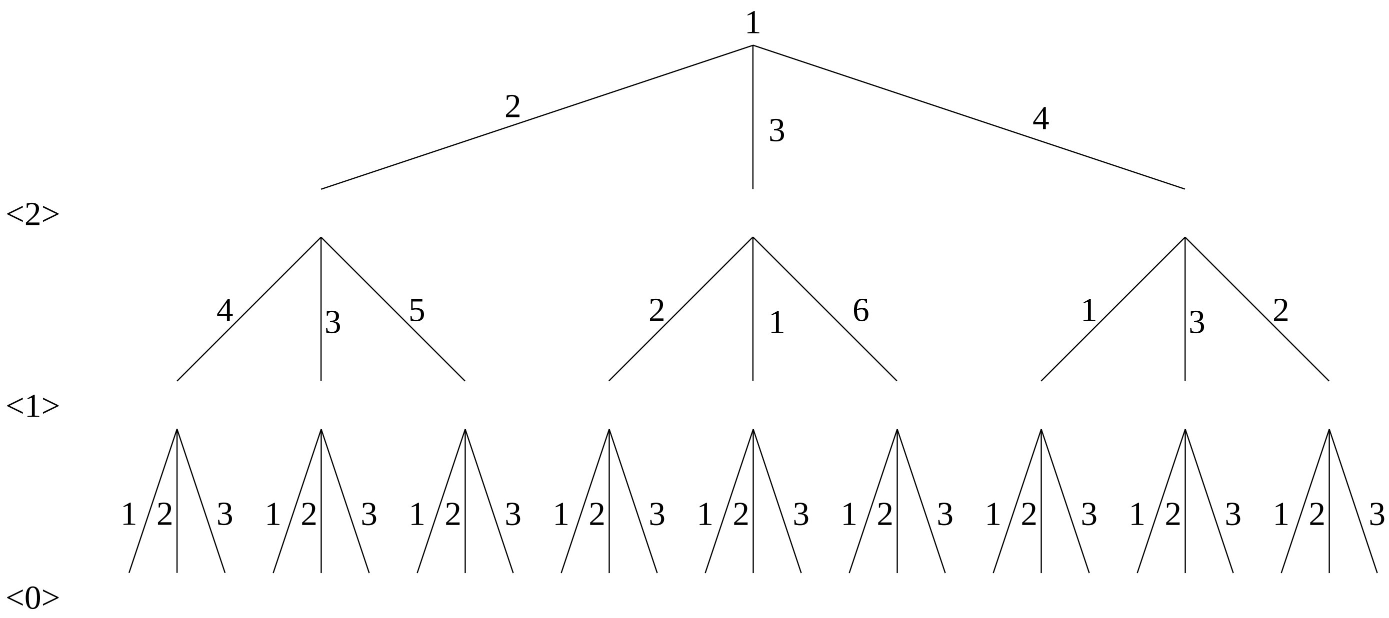
Identical to DFS except that at each point we remove from the frontier the node with the **lowest cost** (i.e., shortest schedule) so far.



A* Search

This is an extension of best-first search that incorporates an under-estimate of the distance remaining to the goal. For example, best-first search wasted time expanding other nodes of cost 5 which obviously were going to cost more after they were expanded.

Suppose we assume that each operator application will cost at least 1 unit and that all solutions are at depth 3. Then we get the following search graph:



A Cost-To-Goal Estimate for Scheduling

We want to estimate how much longer the schedule will need to grow in order to be completed.

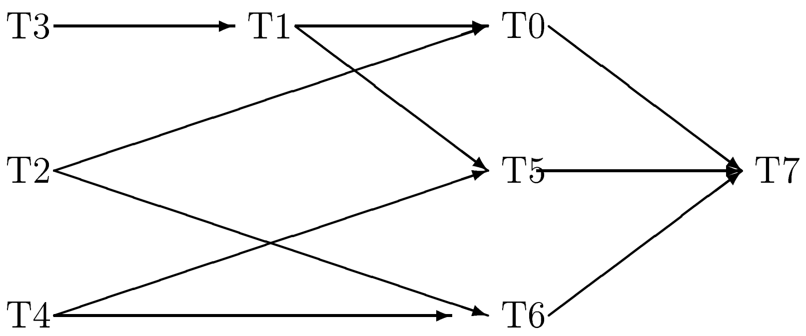
For each unscheduled task compute its earliest possible start time (based on its prerequisite tasks) + its critical-path margin.

The estimated cost-to-goal is the maximum of these values (minus the current schedule length).

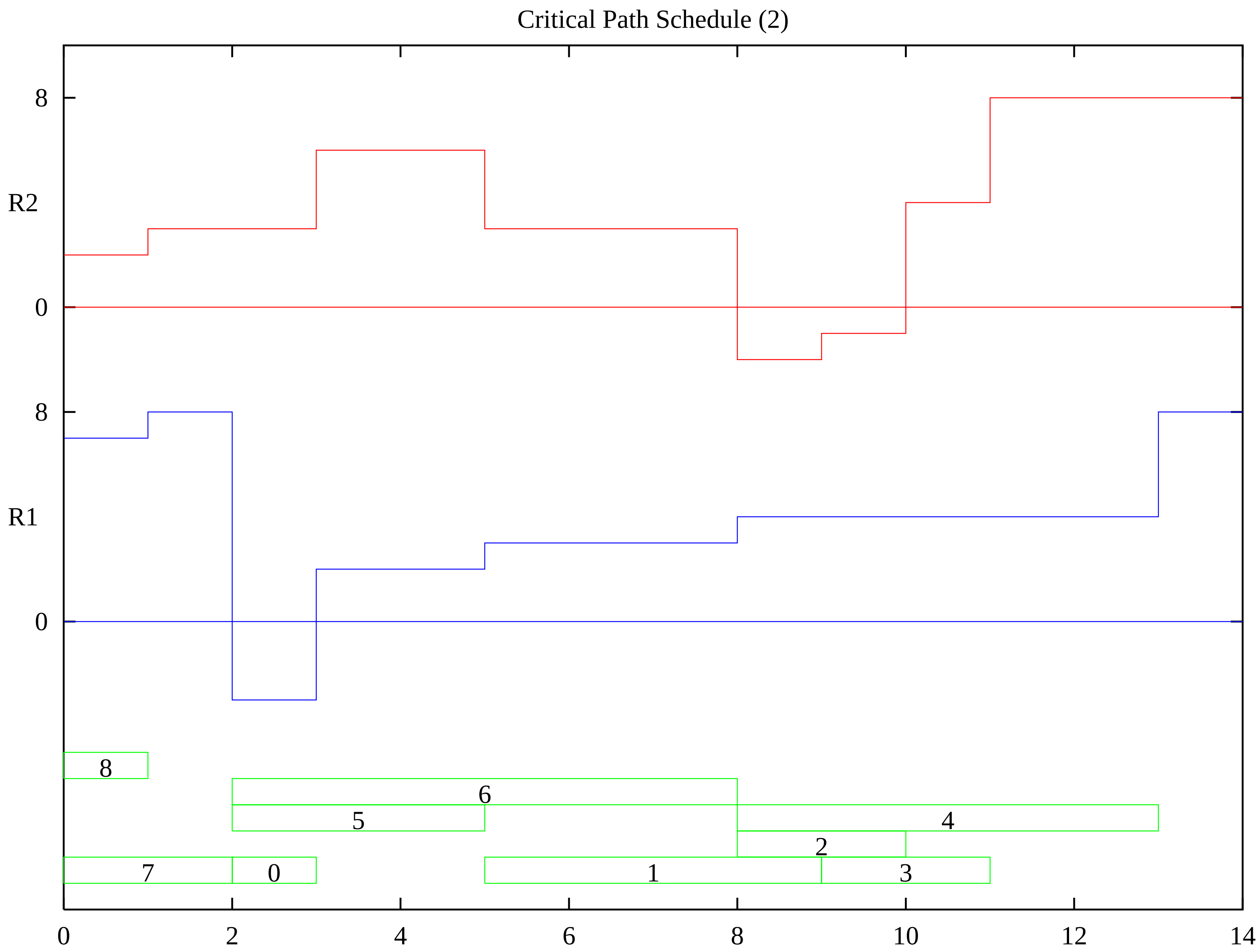
A Simple Scheduling Problem

Task	Duration	After	Resource Type R1	Resource Type R2
T0	1	T1, T2	5	3
T1	4	T3		5
T2	2			5
T3	2			4
T4	5		4	
T5	3	T1, T4	1	2
T6	6	T2, T4	5	
T7	2	T0, T5, T6		5

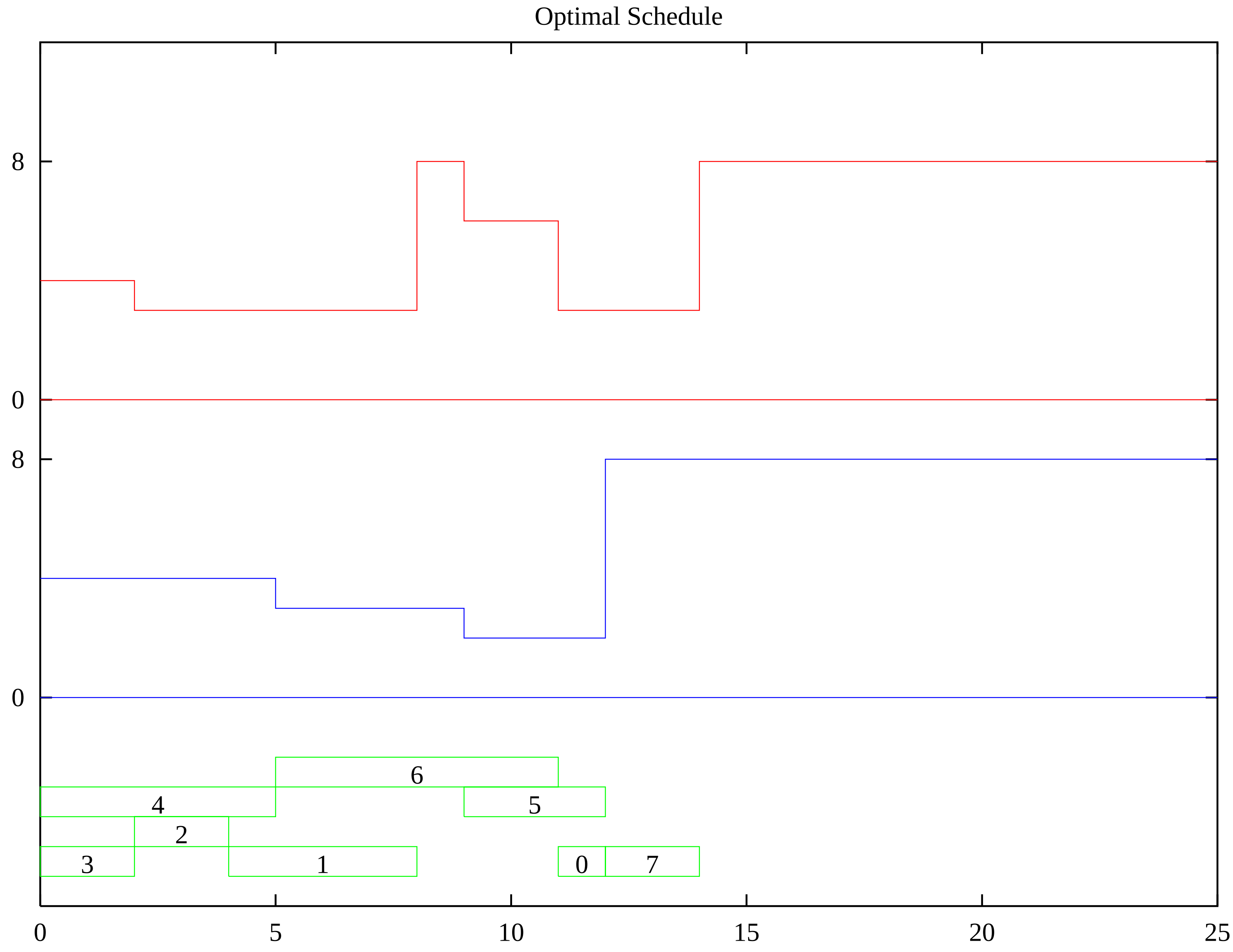
Type	Capacity
R1	8
R2	8



Critical Path Schedule



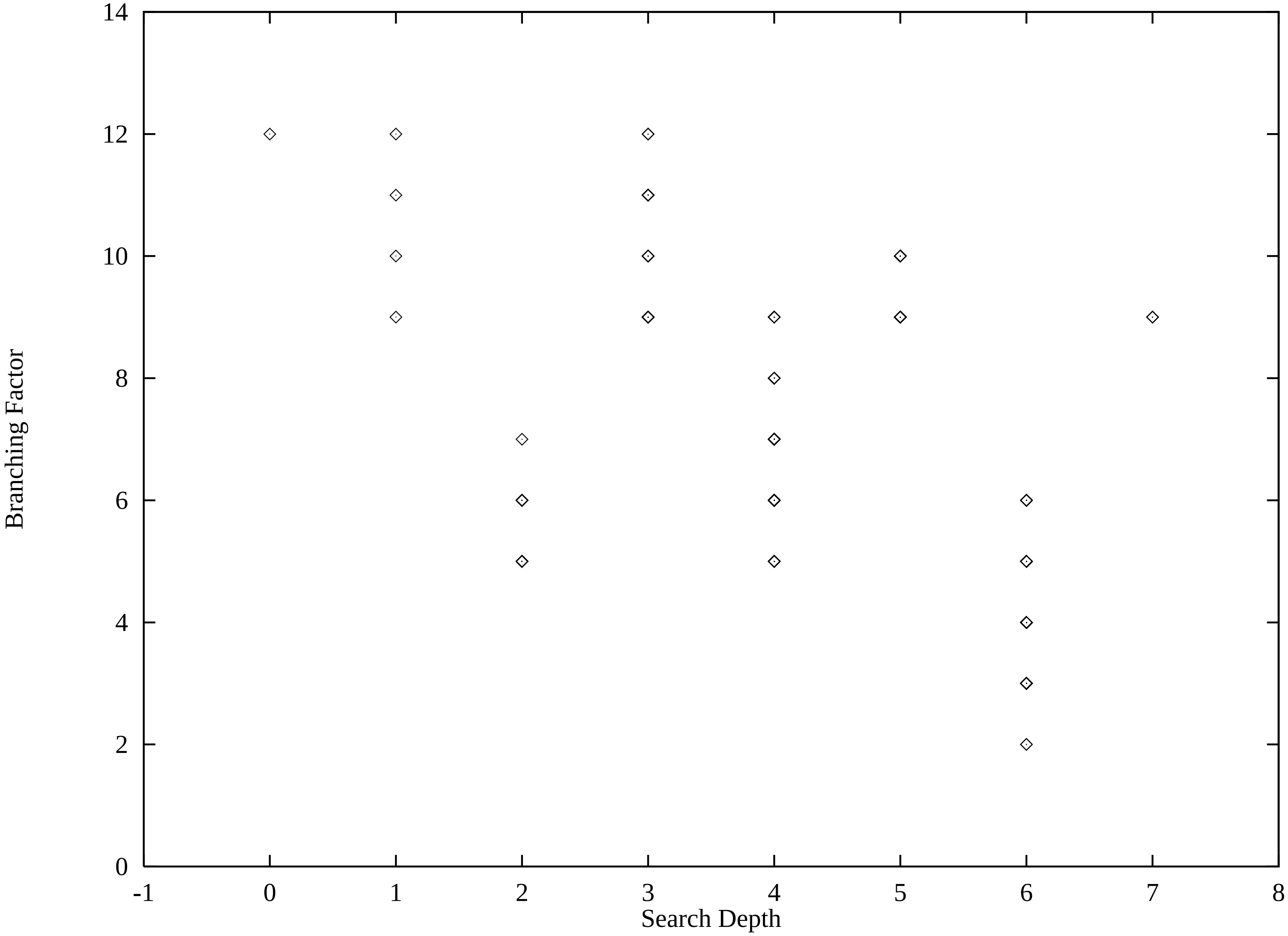
Optimal Schedule

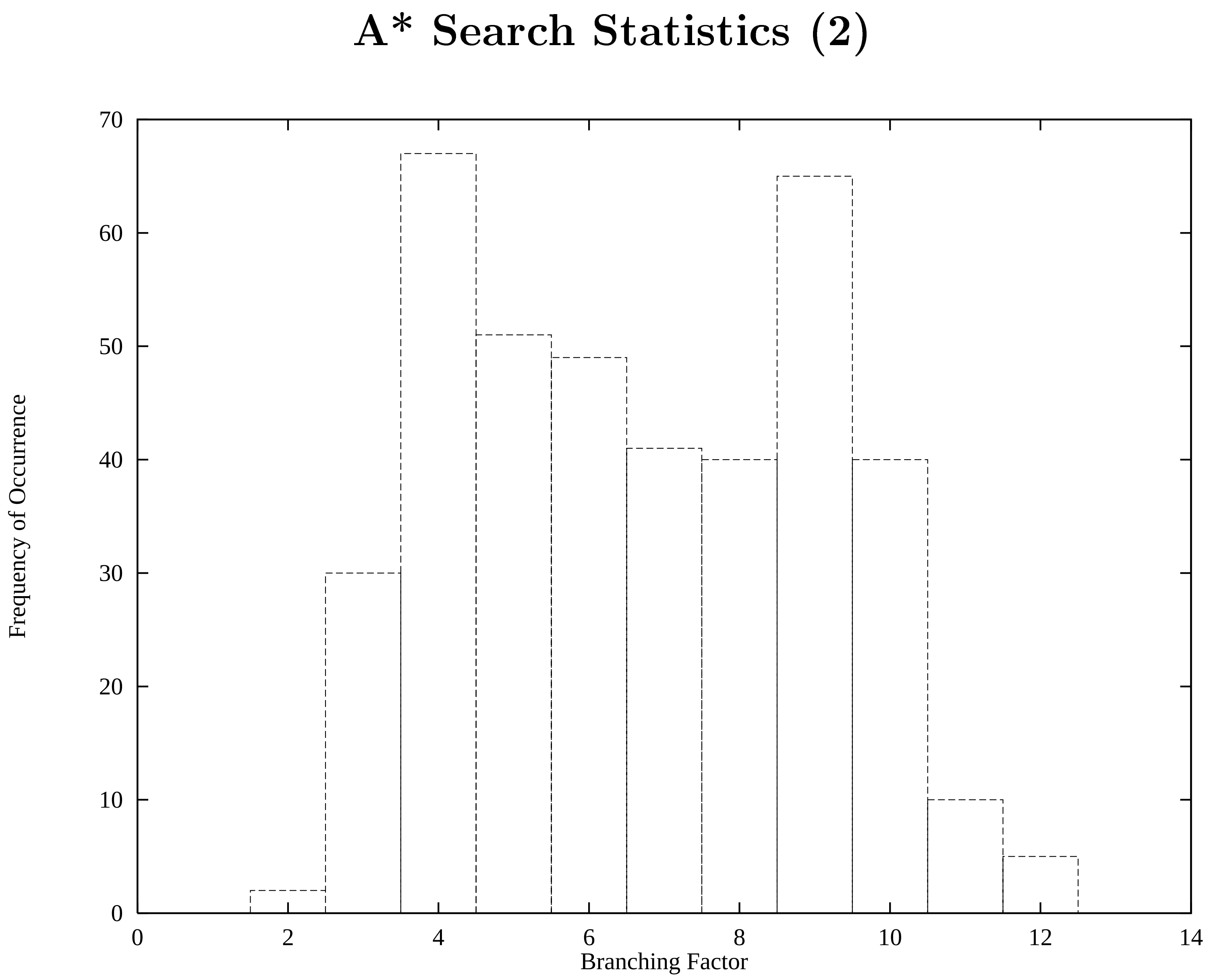


A* Search Statistics

400 nodes expanded

2,823 nodes created




A* Search Statistics (2)
Frequency of Occurrence
Branching Factor
0
10
20
30
40
50
60
70
0
2
4
6
8
10
12
14

A* on NASA Problem

After expanding 1,893 nodes (and creating 51,972 states), 128Mb swap space was exhausted on my Sparcstation 5.

No more than 6 tasks had yet been scheduled.

On the average, each expansion produced 27.5 child nodes (maximum 147 minimum 0, median 19).

Greedy Search

Exhaustive searches are infeasible for larger problems. Some kind of approximate search is required.

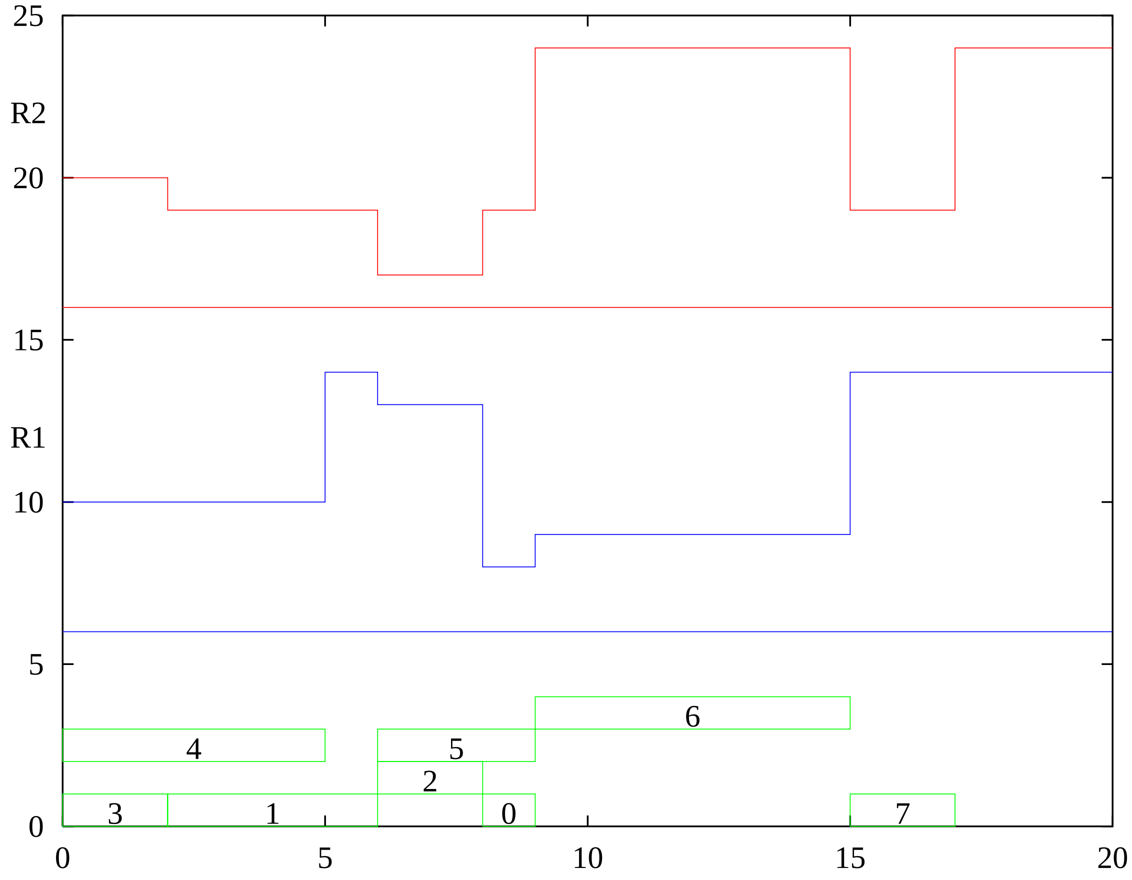
A very cheap approximate search algorithm is greedy search.

```
current = start state;
while (current is not a feasible solution) {
  for each child of current {
     evaluate QUALITY(child);
     current = best child;
     }
  }
```

Greedy search rarely leads to optimal solutions. But it often works fairly well.

Greedy Search Heuristic

One greedy search heuristic is to prefer the child schedule with the shortest schedule length.



This clearly does not produce an optimal schedule, but it is very cheap. The number of nodes evaluated is $B \cdot D$ instead of B^D.

Dispatch Rules

Greedy search heuristics are also sometimes called **dispatch rules**. The heuristic is essentially used by a "dispatcher" to choose which task to schedule next.

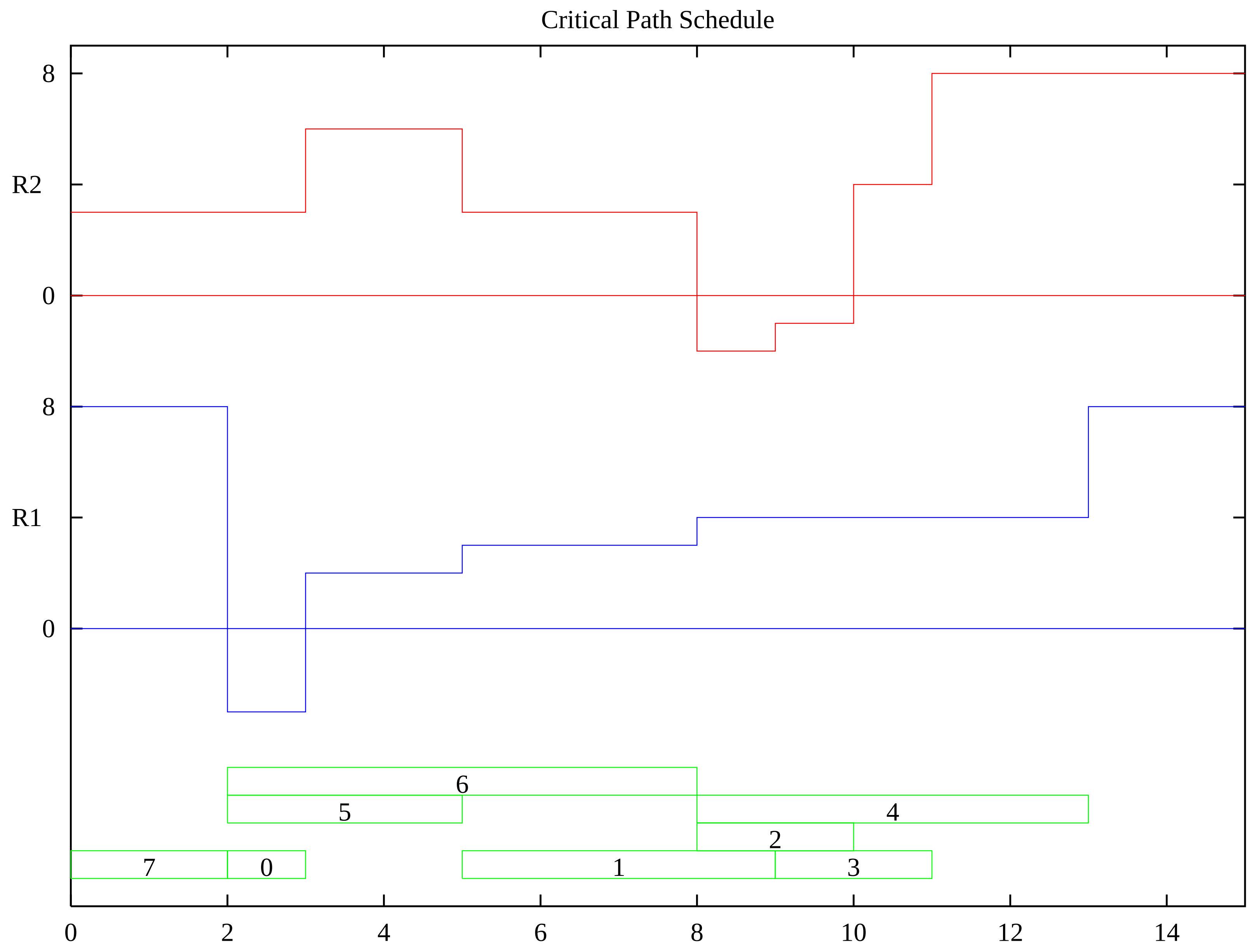
Optimal scheduling is an NP-Complete problem. Most computer scientists believe that this means any algorithm for optimal scheduling must require time exponential in the size of the scheduling problem.

An Alternative Problem Space for Scheduling

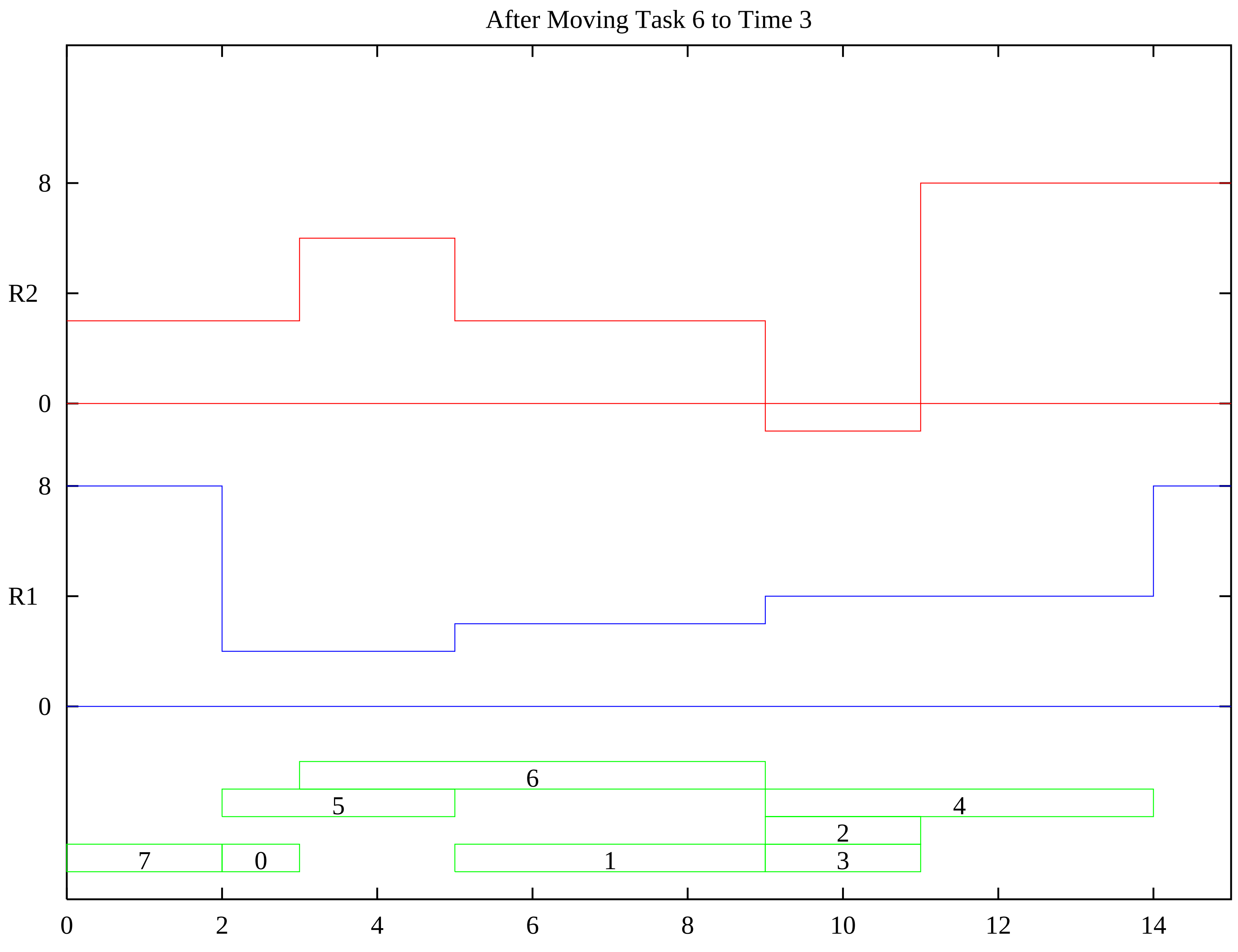
A difficulty with our state space search is that we can't "look ahead" to see what problems will be created by our search decisions. A different kind of problem space—a **repair-based problem space**—makes this easier.

- **state:** a complete schedule in which all tasks have been assigned start times and all prerequisite constraints are satisfied but resource constraints may be violated.
- **start state**: A critical path schedule.
- **operators:** choose a task involved in the earliest constraint violation and move it to the next time at which all of its resources are available. Move all of the postrequisites of the chosen task (recursively) and schedule them as early as their prerequisites permit.
- **goal:** the shortest complete schedule with no overallocated resources.

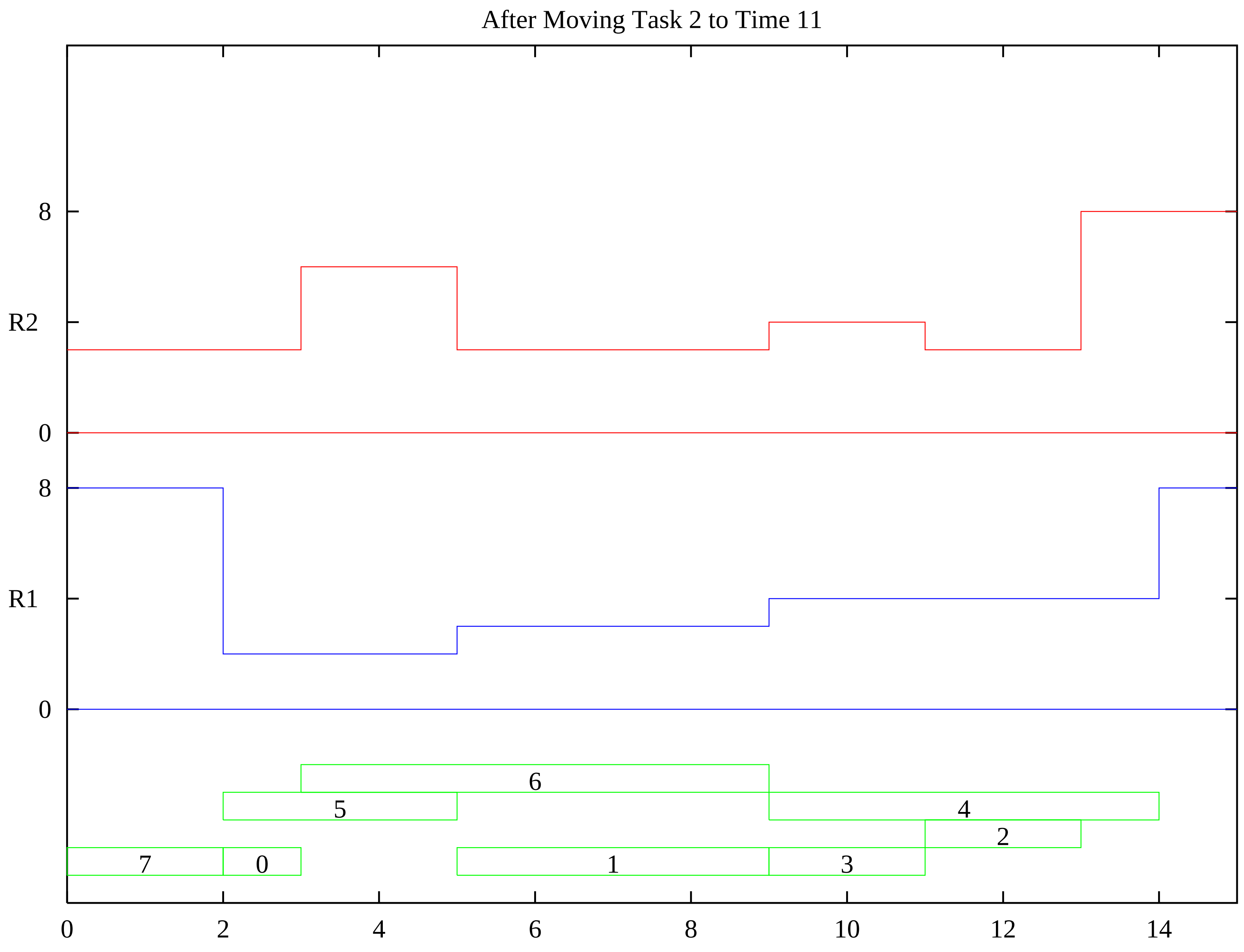
Example of a Move Operation



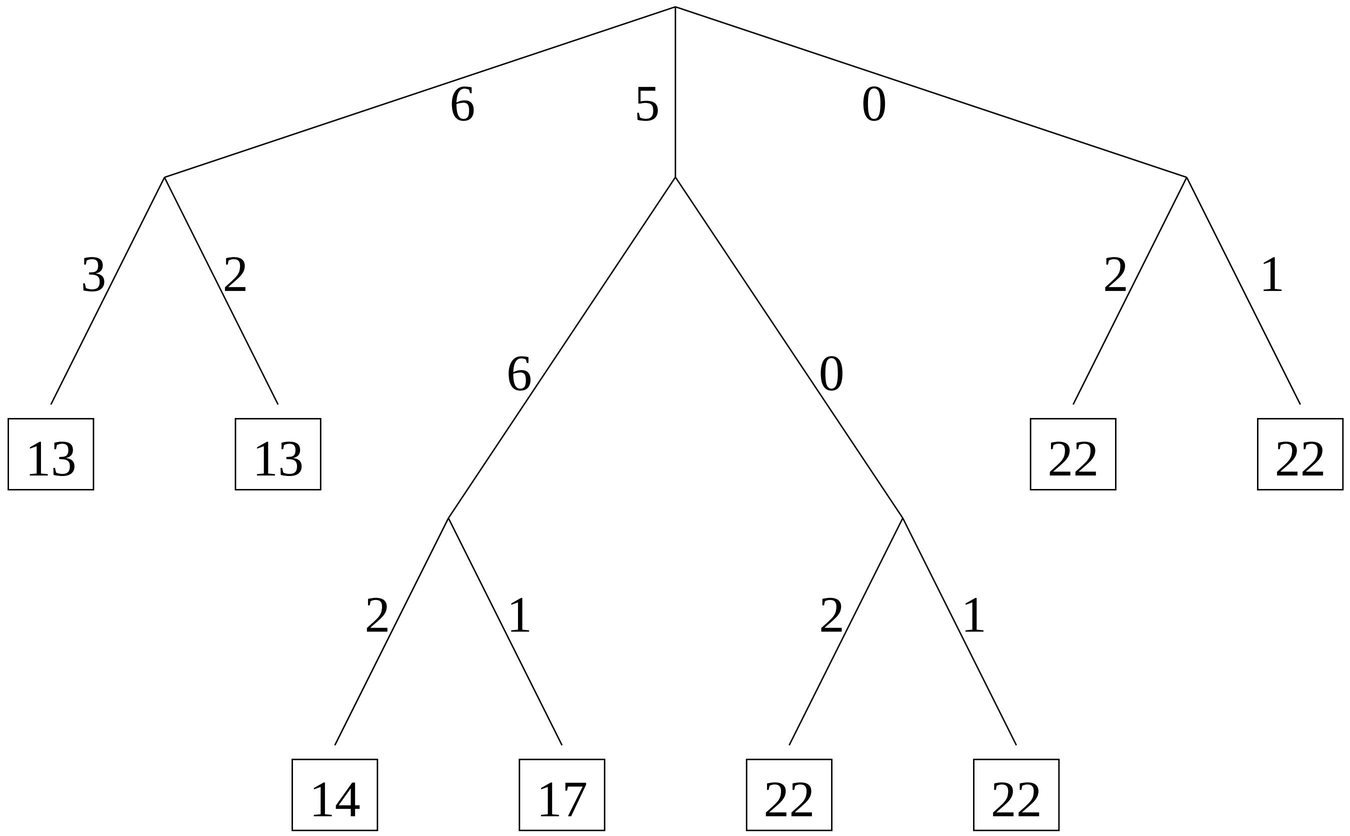
After First Repair



After Second Repair



The Repair Problem Space is Much Smaller

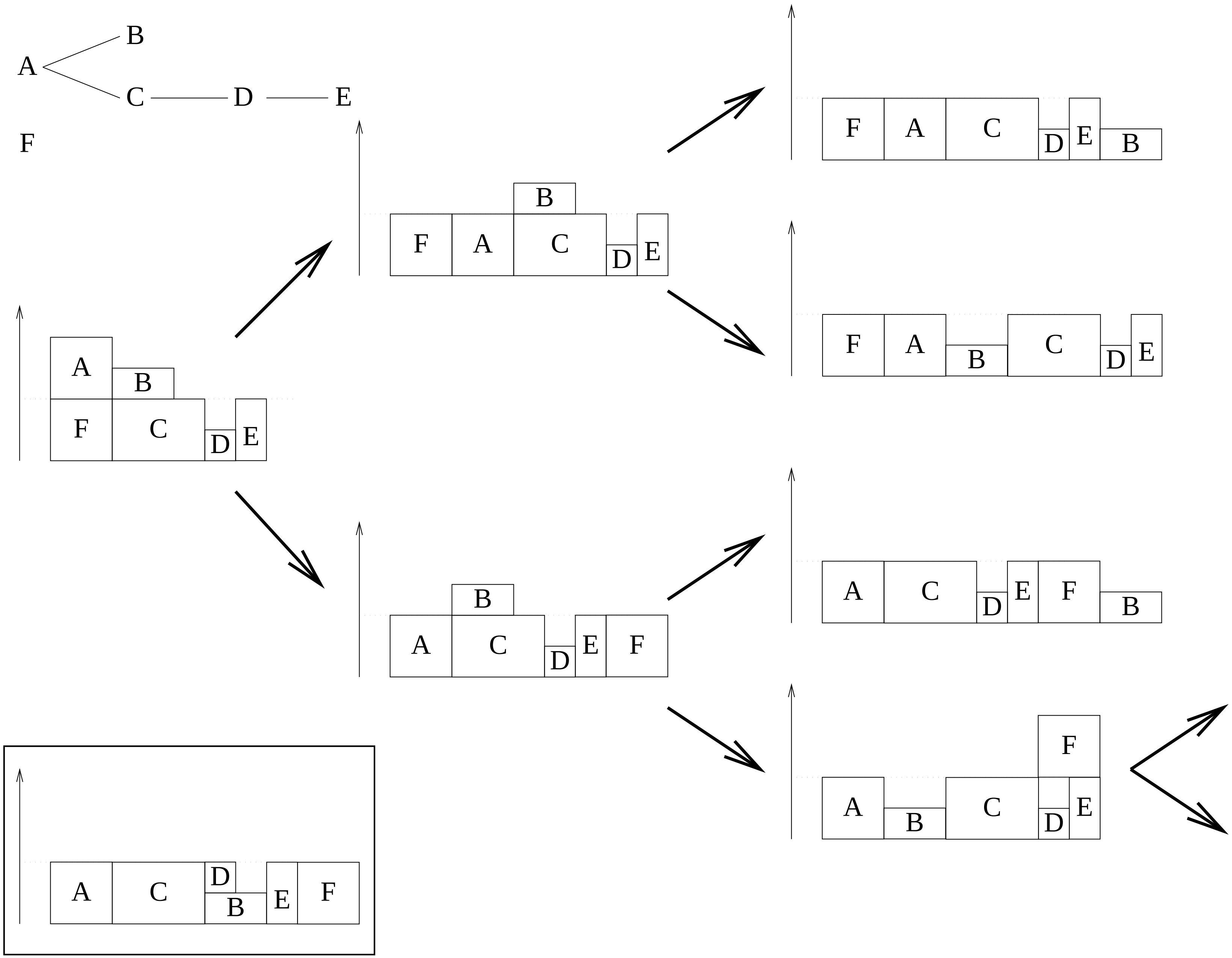


For the example NASA problem, the entire search space contains only 1,942 states.

Problem Space Properties of the Repair Space (NASA example)

- **Branching Factor**. Mean 2.7 (median 3).
- **Solution Depth**. 9.22 (median 9). (No infinite descending paths.)
- **Solution Density**. All solutions are feasible. There are 1,222 feasible solution nodes of which 48 (3.93%) are optimal.
- **Redundancy**. Very little.
- **Completeness**. The space is not complete.

An Example of Incompleteness



Search Algorithms for the Repair Space

All of the search algorithms we described above can be applied to this problem space. I used depth-first search to compute the statistics given above.

There are some search algorithms that are well-suited to repair-based problem spaces. Two important algorithms are **Boltzmann exploration** and **Simulated Annealing**. Both of these are randomized algorithms.

Boltzmann Exploration is appropriate for problem spaces with no cycles (e.g., no inverse operators).

Simulated Annealing is appropriate for problems spaces that contain cycles (e.g., every operator has an inverse).

Randomized Search Algorithms: Boltzmann Exploration

```
temperature = 20.0;
CurrentState = start state;
while (!feasible(CurrentState)) {
   children = expand(CurrentState);
   for each child, let prio(child) = exp(score(child) / temperature);
   totalPrio = sum of prio(child) for each child
   choose a child at random such that each child has probability
      P(child) = prio(child) / totalPrio
   CurrentState = chosen child;
}
```

Choose the next state randomly according to:

$$P(c) = \frac{exp(score(c)/T)}{\Sigma_c \, exp(score(c)/T)}$$

Where c is a child of the current state.

If $T = \infty$, then this becomes simple random search.

If $T = 0$, then this become simple greedy search (choose child with highest score).

In between, this is random search with a bias toward children with higher scores.

Implementing the Random Choice

```
random rng(27384);   // random number generator with seed 27384
float temperature = 50.0;
list<State> PossibleChildren = expand(CurrentState);
listIterator<State> itr(PossibleChildren);
list<float> ChildrenScores;
float totalScore = 0.0;
for (itr.init(); !itr; ++itr) {
  thisScore = exp(score(itr()) / temperature);
  totalScore += thisScore;
  ChildrenScores.add(totalScore);
}
ChildrenScores.reverse();
```

Implementing the Random Choice (2)

```
listIterator<float> scoreitr(ChildrenScores);
float randomNumber = rng.between(0.0, totalScore);
for (itr.init(), scoreitr.init();
      !itr;
      ++itr, ++scoreitr) {
   if (randomNumber <= scoreitr()) {
     CurentState = itr();
     break;
   }
}
```

In practice, as T gets small, we need to avoid overflow of $exp(x)$. We can do this as follows: If $T < 0.01$, then switch to a greedy algorithm.

In scheduling, we want to minimize the score rather than maximize. We can simply negate the scores:

$$exp(-score(c)/T)$$

.

Randomized Search Algorithms: Simulated Annealing

```
CurrentState = Critical path schedule;
CurrentScore = score(CurrentState);
temperature = 50.0;
while (temperature > 0) {
  PossibleChildren = expand(CurrentState);
  ProposedState = randomly choose one of the PossibleChildren;
  ProposedScore = score(ProposedState);
  DeltaScore = ProposedScore − CurrentScore;
  if ((DeltaScore ≤ 0) ||
      (Random(0.0,1.0)) < exp(−(DeltaScore/temperature))) {
        CurrentState = ProposedState;
        CurrentScore = ProposedScore;
        }
  temperature = AnnealingPolicy(temperature);
  }
```

Chemical Analogy for Simulated Annealing

To create a crystal (i.e., a solid with minimum internal energy), you heat the material in liquid form and then cool it slowly ("anneal" it).

As the solution cools, it is constantly bouncing from one state to another according to the Boltzmann distribution

$$P(S) = e^{-\frac{k\Delta E}{T}}$$

where k is a constant, ΔE is the difference between the current energy and the global minimum energy, and T is the current temperature in degrees Kelvin.

The solution may jump to states of higher energy temporarily, but if the temperature is cooled slowly enough, the system will settle into the global minimum energy solution.

Annealing Policies

A wide range of annealing policies have been studied. The simplest policy just decreases the temperature by a fixed amount after every N iterations (e.g. 100).

Other policies decrease the temperature more when progress is made.

A state-of-the-art implementation is available over the network from Lester Ingber `http://www.ingber.com/`.

Boltzmann Exploration for Scheduling

Use the repair-based problem space.

Hold the temperature constant.

Do many repeated runs (as many as time permits) from the starting state until a feasible solution is found. Keep the best solution.

This is essentially an intermediate point between random search and a greedy algorithm.

Overview of C++ code for scheduling

- `ResourceGraph`
  ```
  unsigned int  amount;
  list<ResourceInterval>  Intervals;
  ```
- `ResourceInterval`
  ```
  unsigned int  start;
  unsigned int  duration;
  int  amount;
  ```
- `Task`
  ```
  int                      StartTime;        // < 0 means not assigned
  unsigned int             Duration;
  vector<int>              Resources;        // resource requirements
  list<unsigned int>       Prerequisites;    // indexes into Tasks for prerequisite tasks
  list<unsigned int>       Postrequisites;   // indexes into Tasks for postrequisite tasks
  ```
- `Schedule`
  ```
  vector<Task *>           Tasks;            // the tasks in the schedule
  vector<ResourceGraph *>  Resources;        // the available resources of each kind
  ```

Important Member Functions for Each Class

- `ResourceGraph`

  void Allocate(unsigned int amount, unsigned int start, unsigned int duration);

  void Release(unsigned int amount, unsigned int start, unsigned int duration);

  int FindNext(unsigned int amount, unsigned int start, unsigned int duration);

  int findFirstViolation(int & start, int & duration);

- **Schedule**

  virtual list<Schedule *> startingStates();

  virtual list<Schedule *> expand();

  virtual int isTerminalState();

  void scheduleTask(Task * task, unsigned int start);

  void unschedule();

  int FindNext(Task * task, int start, int initialize = 1);

  void topologicalSort();

Virtual Member Functions and Inheritance

Schedule

```
vector<Task *> Tasks;
vector<ResourceGraph *> Resources;
virtual list<Schedule *> startingStates();
virtual list<Schedule *> expand()
virtual int isTerminalState();
```

ConstructiveSchedule

```
virtual list<Schedule *> startingStates();
virtual list<Schedule *> expand()
virtual int isTerminalState();
```

IRSchedule

```
virtual list<Schedule *> startingStates();
virtual list<Schedule *> expand()
virtual int isTerminalState();
```

Generic Random Search Algorithm

```
IRSchedule Original;
list<Schedule *> frontier = Original.startingStates();
Schedule * Current = 0;
while (1) {
  int nchildren = frontier.length();
  if (nchildren ≤ 0) break;
  int ichild = RandomGenerator.between(0, nchildren − 1);
  if (Current) delete Current;
  Current = frontier[ichild];  // choose a random child
  listIterator<Schedule *>  itr(frontier);
  for (itr.init(); !itr; ++itr) {     // release memory of other children
    if (itr() ≠ Current) delete itr();
  }
  if (Current→isTerminalState()) break;  // quit if terminal
  frontier = Current→expand();        // else expand Current
  frontier.reverse();  // make the indexes match the order
}
```

Member Functions for Constructive and Repair Schedules

- `ConstructiveSchedule`
  int fhat(); // *estimate of remaining schedule length*
- `IRSchedule`
  int unscheduleTask(Task * task);

  int unscheduleTaskandDependents(Task * task);

  void moveTask(unsigned int itask);

  int findFirstViolation(int & start, int & duration, int & resindex);

  list<unsigned int> conflictingTasks(int start, int duration, int resindex);

Summary of State-Space Search

- **The hard part is** *formulating the problem space*.

  A good problem space is small (low branching factor, short solution depth, high solution density, no redundancy) and yet complete. Completeness is often sacrificed to get a small space.

- A^* **search guarantees optimal solutions.** But it does not kill the exponential growth of the search space.

- **Greedy Algorithms can find fairly good solutions quickly.** In scheduling, these are called **dispatch rules**.

- **Simulated Annealing** is a randomized version of the greedy algorithm that is best-suited to repair-based problem spaces. It provides a smooth tradeoff between CPU time and solution quality.

Summary of Scheduling

- **Scheduling is a very important problem.** Total enterprise scheduling can provide strategic competitive advantages for corporations.
- **Repair-based problem spaces are much smaller than constructive problem spaces**. In a repair-based space, the search process can focus only on the tasks involved in resource violations. Easy parts of the space are ignored.
- **A simulated-annealing scheduler gives excellent performance on a NASA task.** It is also provides the foundation for a start-up company: Red Pepper Software (now owned by People-Soft).
- **Reinforcement learning methods can learn very good evaluation functions for the greedy algorithm.**