Optimal Crane Scheduling

Ionuţ Aron
Iiro Harjunkoski
John Hooker
Latife Genç Kaya

March 2007

Problem

- Schedule 2 cranes to transfer material between locations in a manufacturing plant.
  - For example, copper processing.
  - Cranes move on a common track.
  - Cranes cannot move past each other.
    - One crane can yield to another.

Problem

Cranes can move vertically and perhaps side to side.

We are concerned with longitudinal movements.

Problem

- Constraints:
  - Some 300 jobs, each with a time window and priority.
  - Precedence relations between jobs.
  - A job may require several stops.
- Objective: minimize total penalty
  - Penalties reflect deviation from desired release times and/or earliest completion times.

Problem

- Three problems in one:
  - Assign jobs to cranes.
  - Find ordering of jobs on each crane.
  - Find space-time trajectory of each crane.
    - Crane scheduling problems are coupled since the cranes must not pass one another.

Relation to EWO

- In practice, the cranes often can't keep up with what seems to be a reasonable production schedule.
  - Is this because a hand-created crane schedule is far from optimal?
  - Or because **any** feasible schedule forces the cranes to lag behind?
- We would like to resolve this issue.
  - We therefore use an **exact method** to seek a feasible trajectory.

Relation to EWO

- Solution of crane problem can suggest a revised production schedule.
  - Spread out the release times.
- May also require revised time windows.
  - Which ones?

Precedence Constraints

- The crane problem has more complex precedence constraints than ordinarily occur in scheduling problems.
  - **Hard** precedence constraints apply to **groups** of jobs.
  - **Soft** precedence constraints are enforced by imposing penalties.

Precedence Constraints

- Hard constraint
  - Let *S*, *T* be sets of jobs.
  - *S* < *T* means that all jobs in *S* must run before any job in *T*.
    - Jobs within *S* may occur in any order, similarly for *T*.
    - {cranes that perform the jobs in *S* } = {cranes that perform the jobs in *T* }
    - Jobs in neither *S* nor *T* can run between *S* and *T*.
  - *S* ≤ *T* means that all jobs in *S* assigned to a given crane must run before any job in *T* assigned to that crane.

Precedence Constraints

- Soft constraint for consecutive jobs
  - Suppose $S < T$ and S, T should occur consecutively.
  - To enforce this:
    - Impose $S < T$.
    - Give jobs in S very similar release times to jobs in T.
    - Impose high penalties on jobs in S, T.

Two-phase Algorithm

- Phase 1: Local search
  - Assign jobs to cranes
  - Sequence jobs on each crane
  - Solve simultaneously by local search.
- Phase 2: Dynamic programming (DP)
  - Find optimal space-time trajectory for the cranes.
  - Solve for the two cranes simultaneously.
  - Would like to be able to find a feasible trajectory if one exists, despite combinatorial nature of problem

Local Search

- Neighborhood is defined by two types of moves.
  - Change assignment
    - Move a job to the other crane
    - Swap two jobs between cranes.
  - Change sequence
    - Swap two jobs.
- Lower bound on penalty
  - Compute penalty if cranes could move directly from one stop to the next.

Local Search

- Begin with a greedy solution.
  - Build it by assigning jobs in the order of release time.
  - Assign job with stops near the left (right) end of the track to the left (right) crane.
  - While maintaining balance between cranes.

Local Search

- In each round, explore a neighborhood of the current solution until a feasible solution is found.
  - Exclude neighbors that violate precedence, assignment constraints.
  - Exclude neighbors who penalty bound is worse than best solution found so far.
  - If DP finds solution infeasible, try other neighbors.

Local Search

- Next round looks at neighborhood of solution obtained in previous round.
  - Or at neighborhood of solution obtained by a swap of the previous solution, if no feasible neighbor was found.

Dynamic Programming

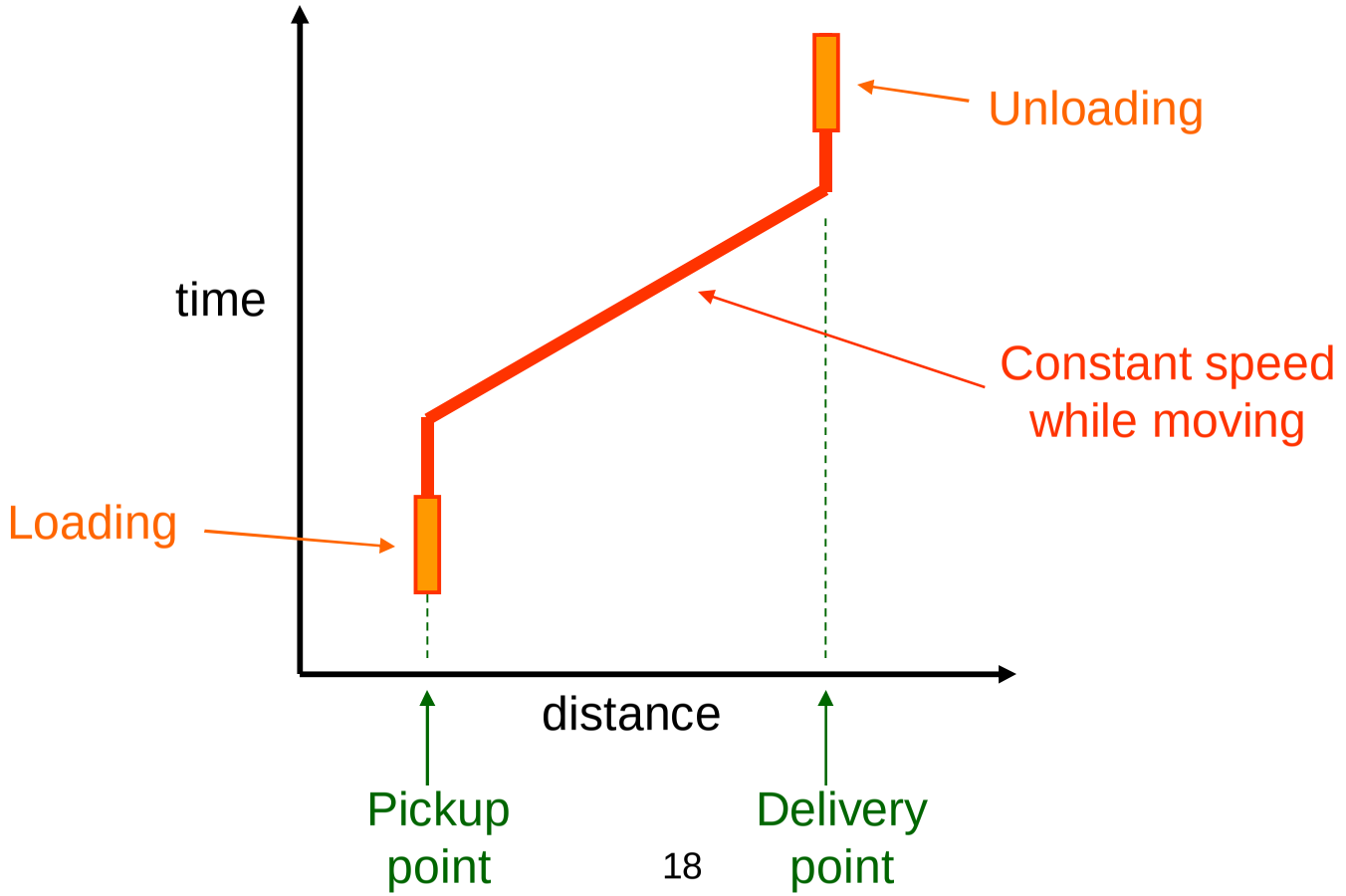
- Each job consists of one or more tasks.
  - Order of tasks within a job is fixed.
  - Each tasks requires that the crane stop to process the task (load, unload, etc.).
- Given for each stop:
  - Location.
  - Processing time.

Dynamic Programming

- Find optimal trajectory for each crane.
  - Sequence of jobs on each crane is given.
  - Minimize sum of penalties, which depend on pickup and delivery times.
- More difficult than finding an optimal space-time path.
  - Must allow for loading/unloading (processing) time.
  - Processing time and location depend on which stop of which job is being processed.
  - Cranes are processing most of the time.

Dynamic Programming

- Space-time trajectory of a crane for two stops.



Dynamic Programming

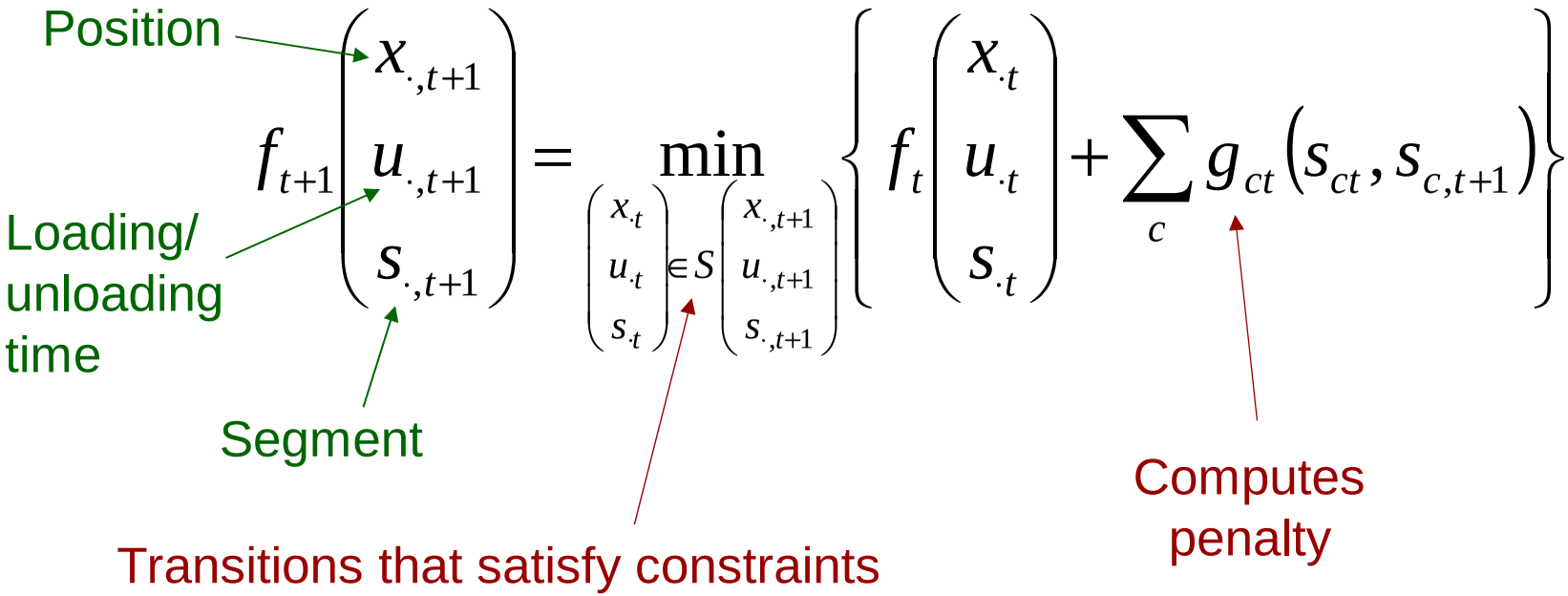
- High resolution needed to track crane motion.
  - Time horizon: several hours
  - Granularity: 10-second intervals
    - Thousands of discrete times
  - Cranes are stationary most of the time
    - While loading/unloading
    - But motion is fast when it occurs (e.g. 1 meter/sec)

Dynamic Programming

- Main issue: size of state space.
- State variables:
  - Position of each crane on track.
  - How long each crane has been loading/unloading.
  - Next (current) stop for each crane

Dynamic Programming

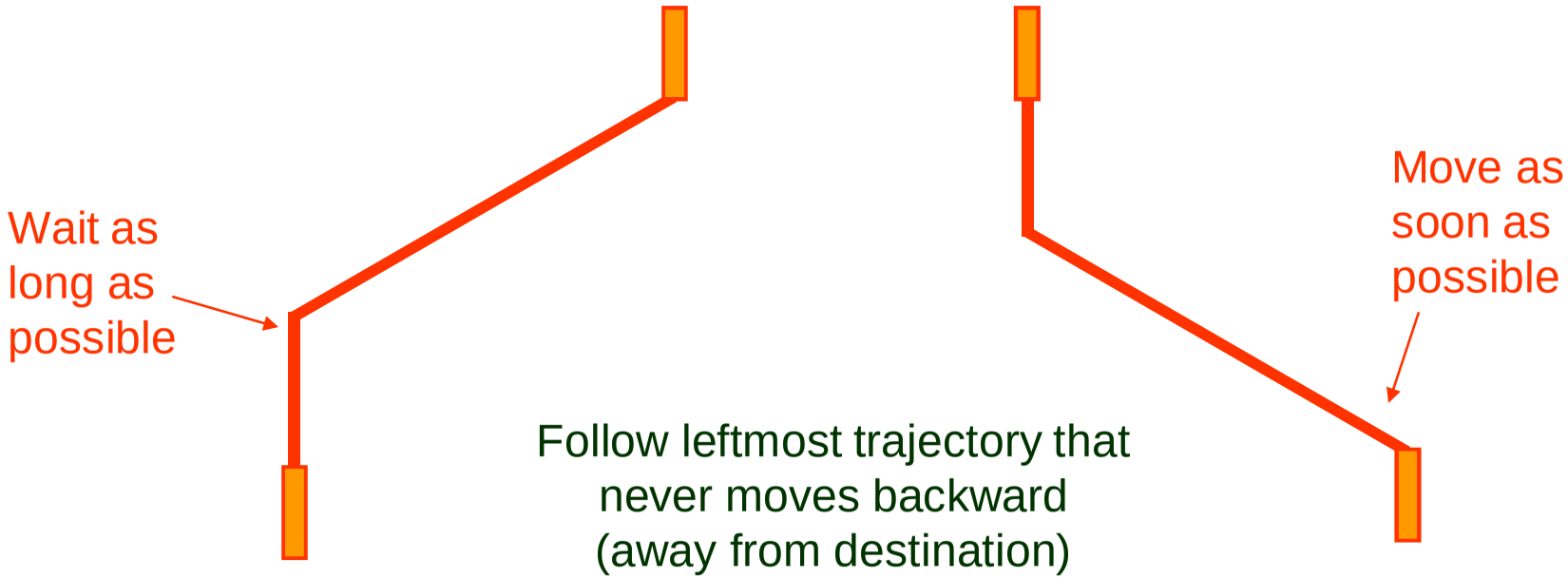
- DP recursion:



$$f_{t+1}\begin{pmatrix} x_{\cdot,t+1} \\ u_{\cdot,t+1} \\ s_{\cdot,t+1} \end{pmatrix} = \min_{\begin{pmatrix} x_{\cdot t} \\ u_{\cdot t} \\ s_{\cdot t} \end{pmatrix} \in S \begin{pmatrix} x_{\cdot,t+1} \\ u_{\cdot,t+1} \\ s_{\cdot,t+1} \end{pmatrix}} \left\{ f_t \begin{pmatrix} x_{\cdot t} \\ u_{\cdot t} \\ s_{\cdot t} \end{pmatrix} + \sum_c g_{ct}\left(s_{ct}, s_{c,t+1}\right) \right\}$$

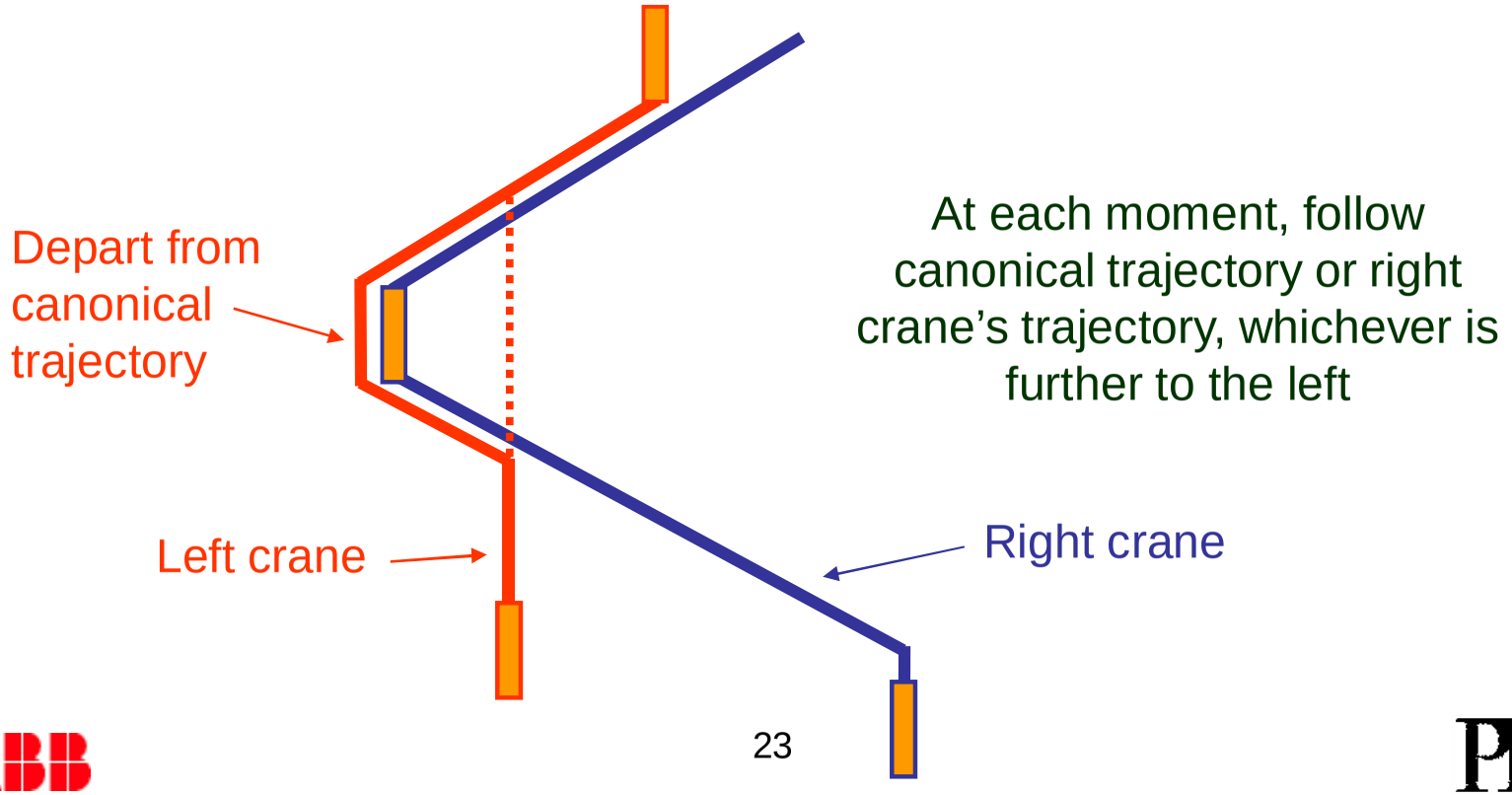
Dynamic Programming

- State space reduction
  - *Canonical* trajectory for the left crane:



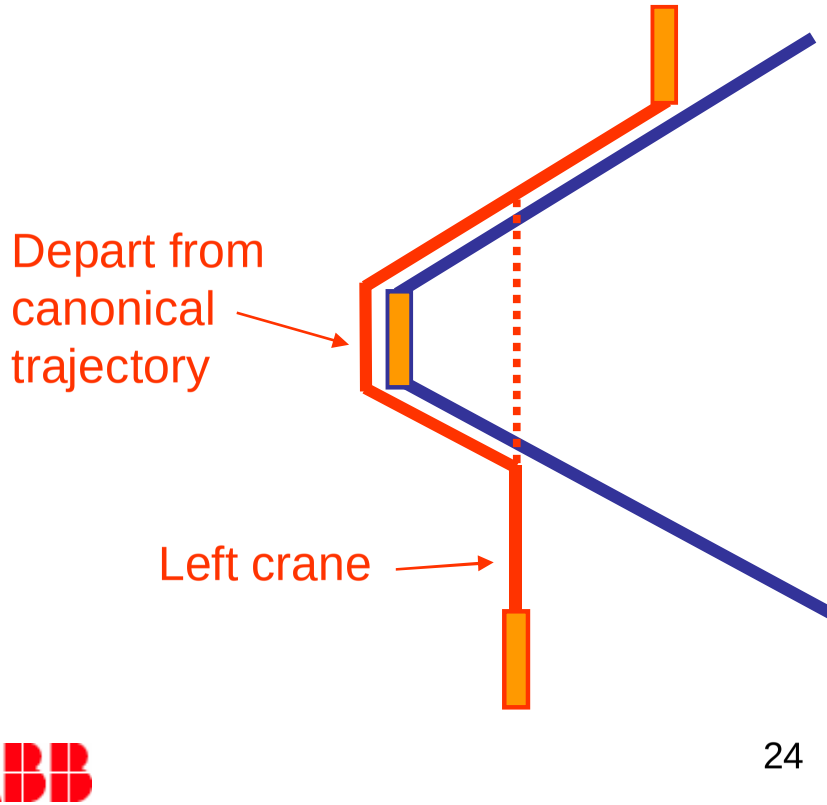
Dynamic Programming

- State space reduction
  - *Minimal* trajectory for left crane:



Dynamic Programming

- State space reduction
  - Properties of the minimal trajectory:

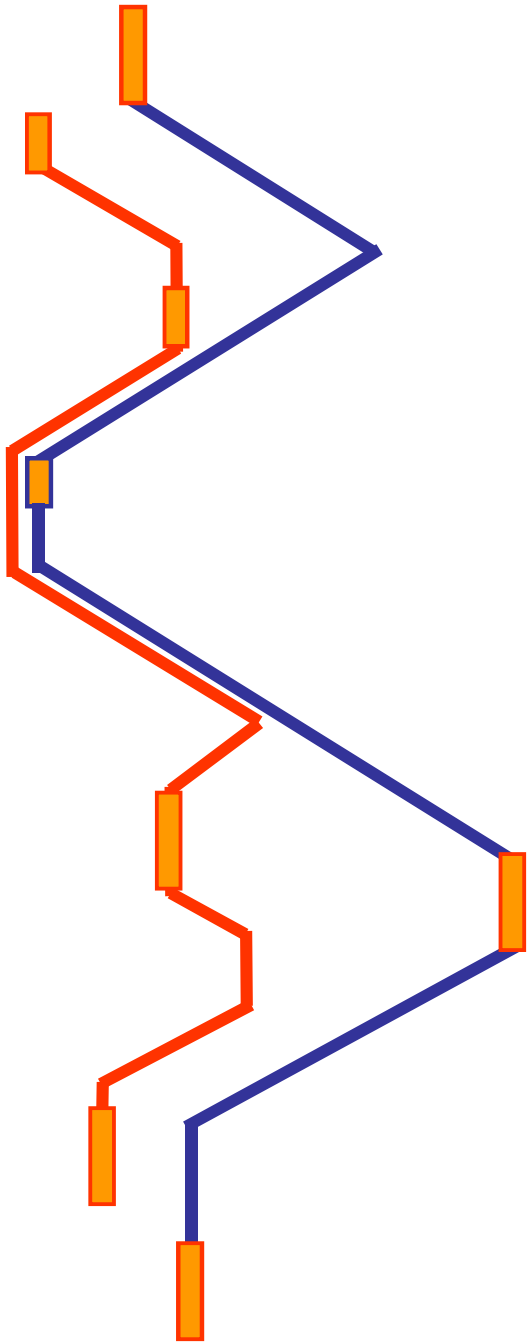


• The left crane never stops *en route* unless it is adjacent to the right crane.

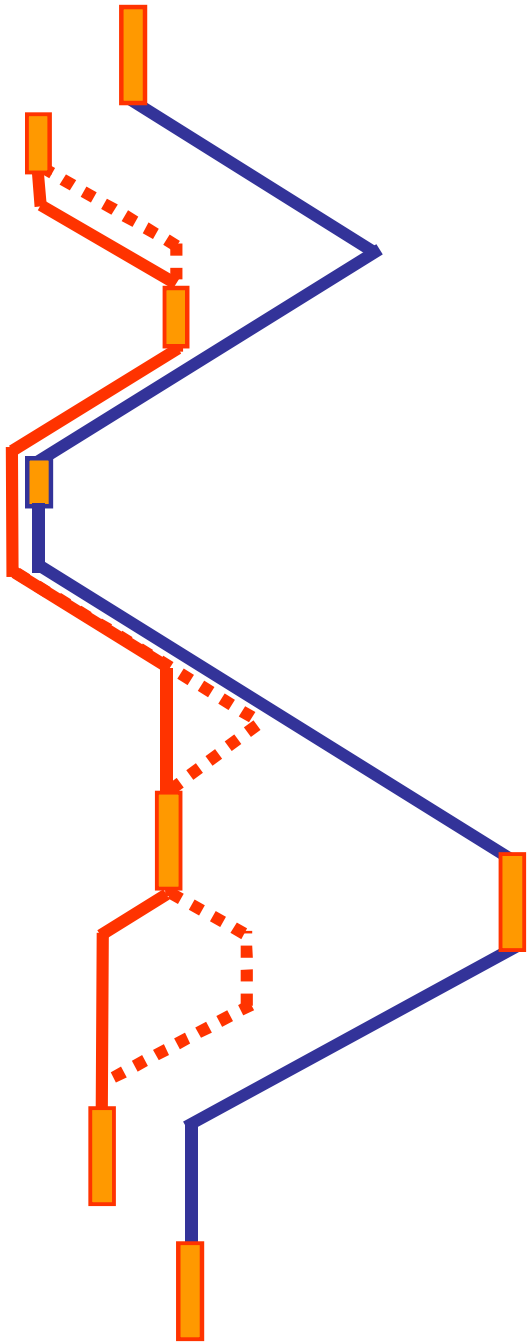
• The left crane never moves backward unless it is adjacent to the right crane.

Dynamic Programming

- State space reduction
  - **Theorem:** Some optimal pair of trajectories are minimal with respect to each other.

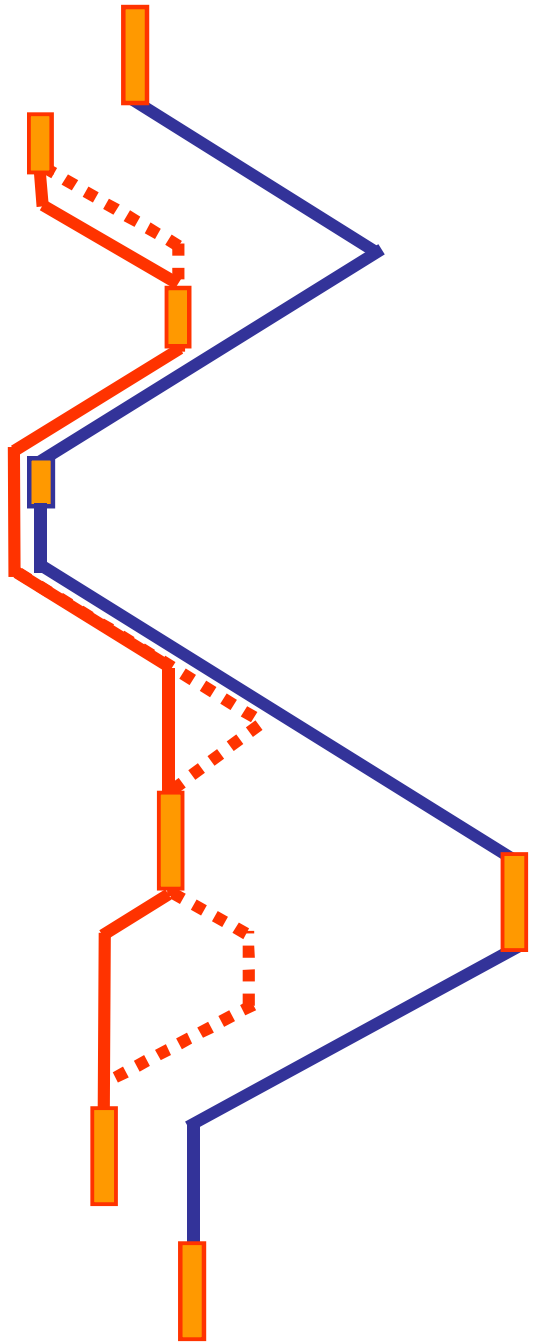


Suppose these segments are part of an optimal trajectory.



Suppose these segments are part of an optimal trajectory.

Change left crane to a minimal trajectory with respect to the right crane

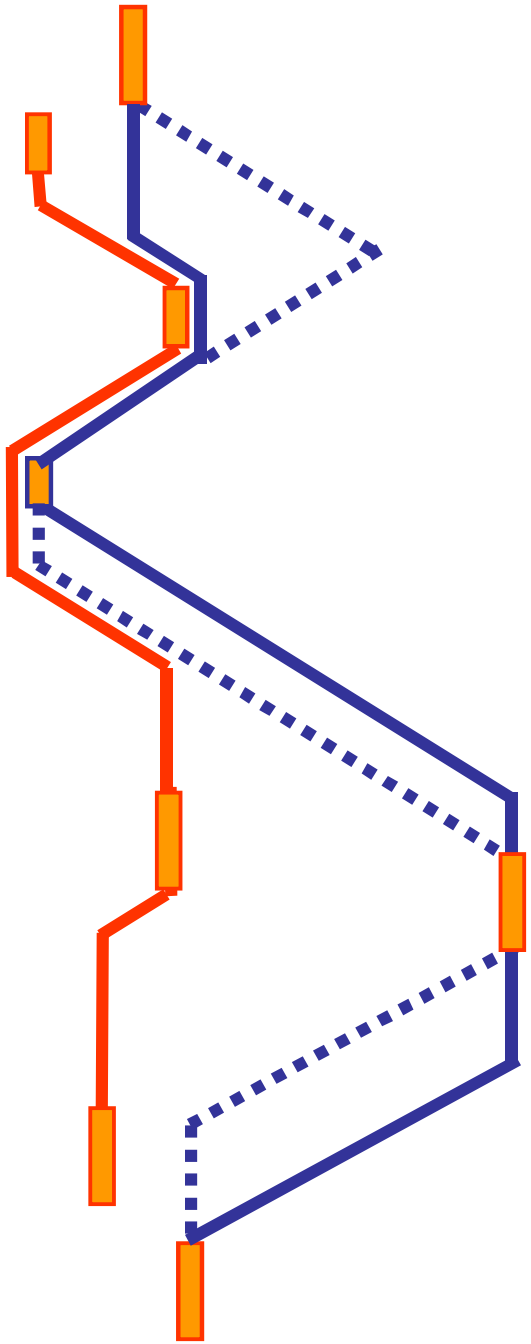


Suppose these segments are part of an optimal trajectory.

Change left crane to a minimal trajectory with respect to the right crane.

This is still feasible, because

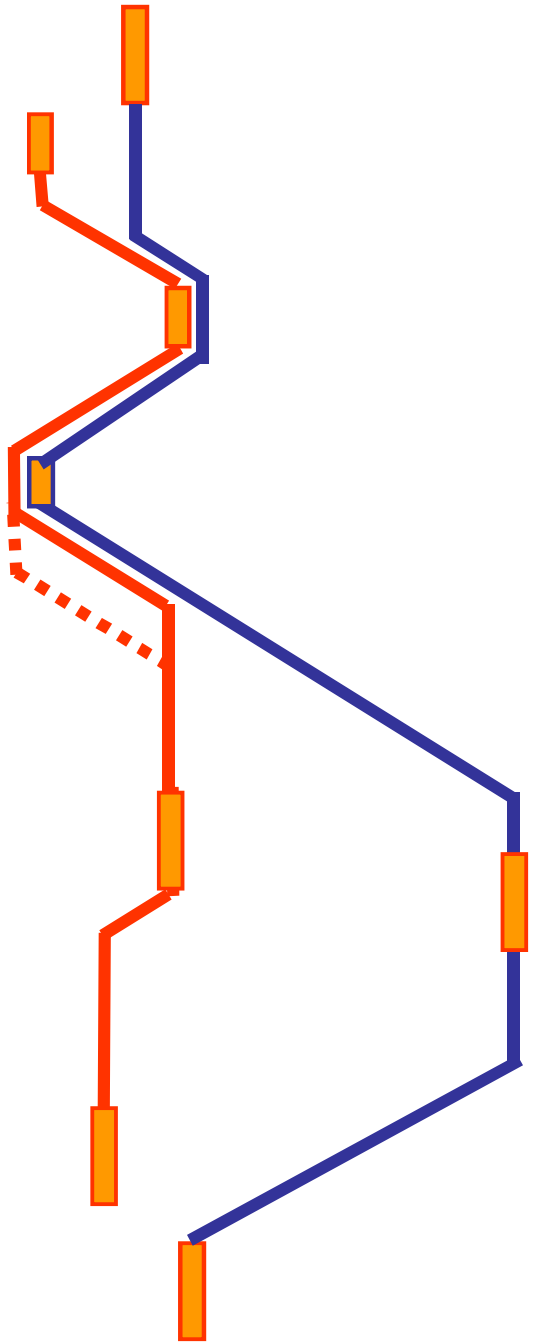
- there is no interference from right crane
- pickup and delivery times are unchanged.



Suppose these segments are part of an optimal trajectory.

Change left crane to a minimal trajectory with respect to the right crane.

Change right crane to a minimal trajectory wrt left crane. Still feasible.

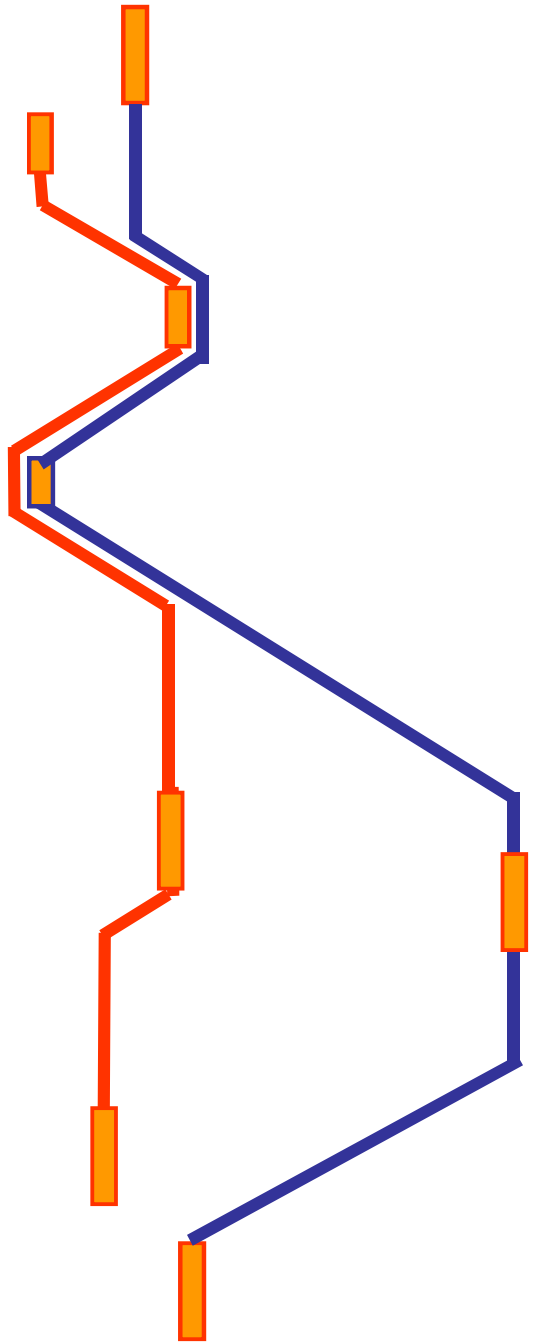


Suppose these segments are part of an optimal trajectory.

Change left crane to a minimal trajectory with respect to the right crane.

Change right crane to a minimal trajectory wrt left crane.

Change left crane to a minimal trajectory wrt right crane. Still feasible.



Suppose these segments are part of an optimal trajectory.

Change left crane to a minimal trajectory with respect to the right crane.

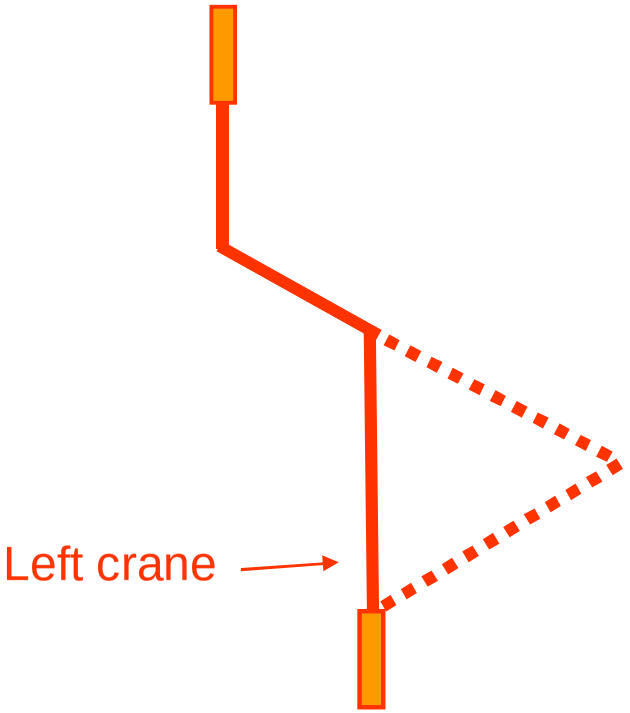
Change right crane to a minimal trajectory wrt left crane.

Change left crane to a minimal trajectory wrt right crane. Still feasible.

This solution is a **fixed point**.

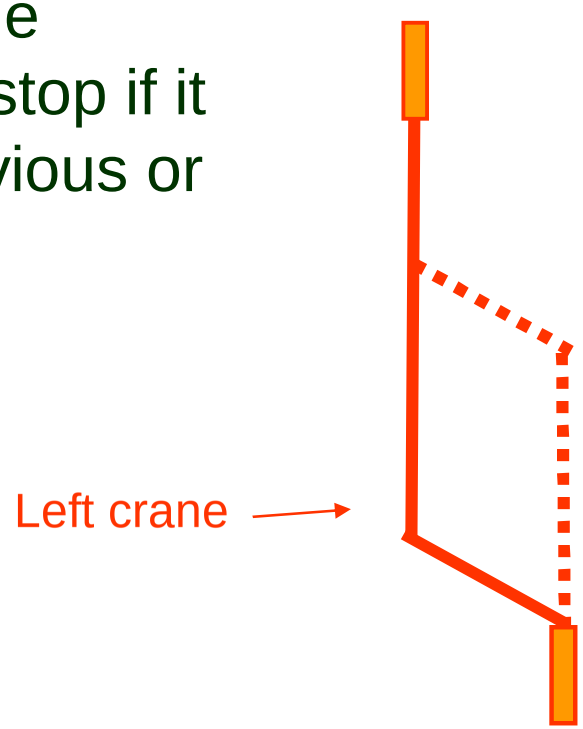
Dynamic Programming

- State space reduction
  - **Corollary:** When not processing, the left crane is never to the right of both the previous and next stops.



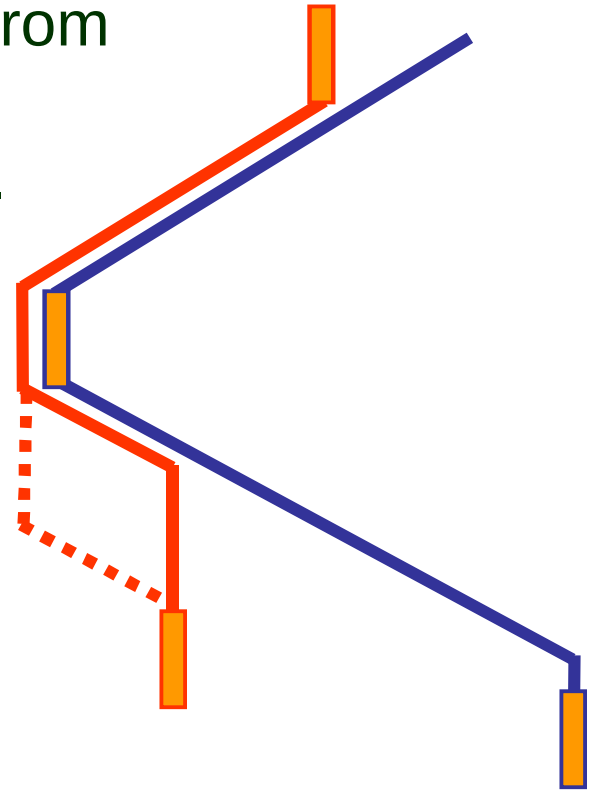
Dynamic Programming

– **Corollary:** When not processing, the left crane moves toward the next stop if it is to the right of the previous or next stop.



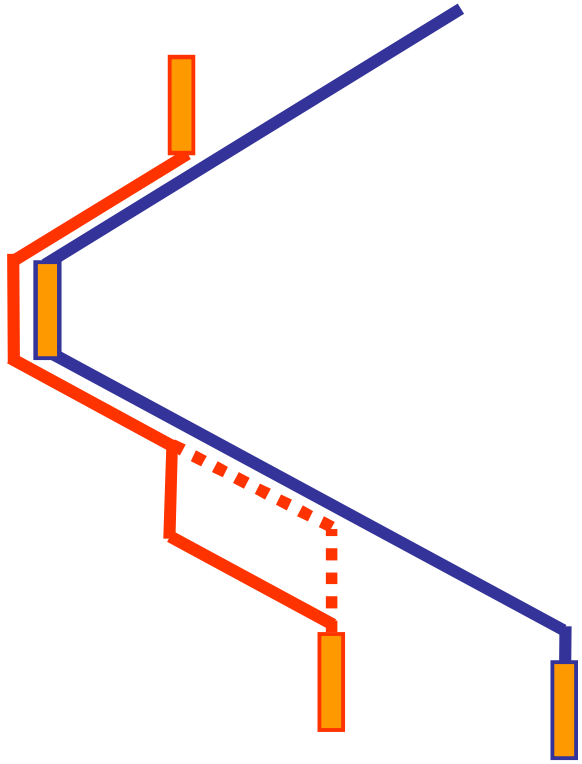
Dynamic Programming

– **Corollary:** a crane never moves in a direction away from the next stop unless it is adjacent to the other crane.



Dynamic Programming

– **Corollary:** When not processing, the left crane is stationary only if it is (a) at the previous or next processing location, whichever is further to the left, or (b) adjacent to the other crane.



Computational Results

- Problem characteristics.
  - 10-60 jobs, 1 to 5 stops per job.
  - 10 crane positions (realistic).
  - Time windows.
  - Objective function: Minimize sum over all jobs of
    - Time lapse between release time and pickup.
    - + time lapse between EFT and delivery.
  - Theoretical maximum number of states in any given stage is approximately 10^8 – 10^9.

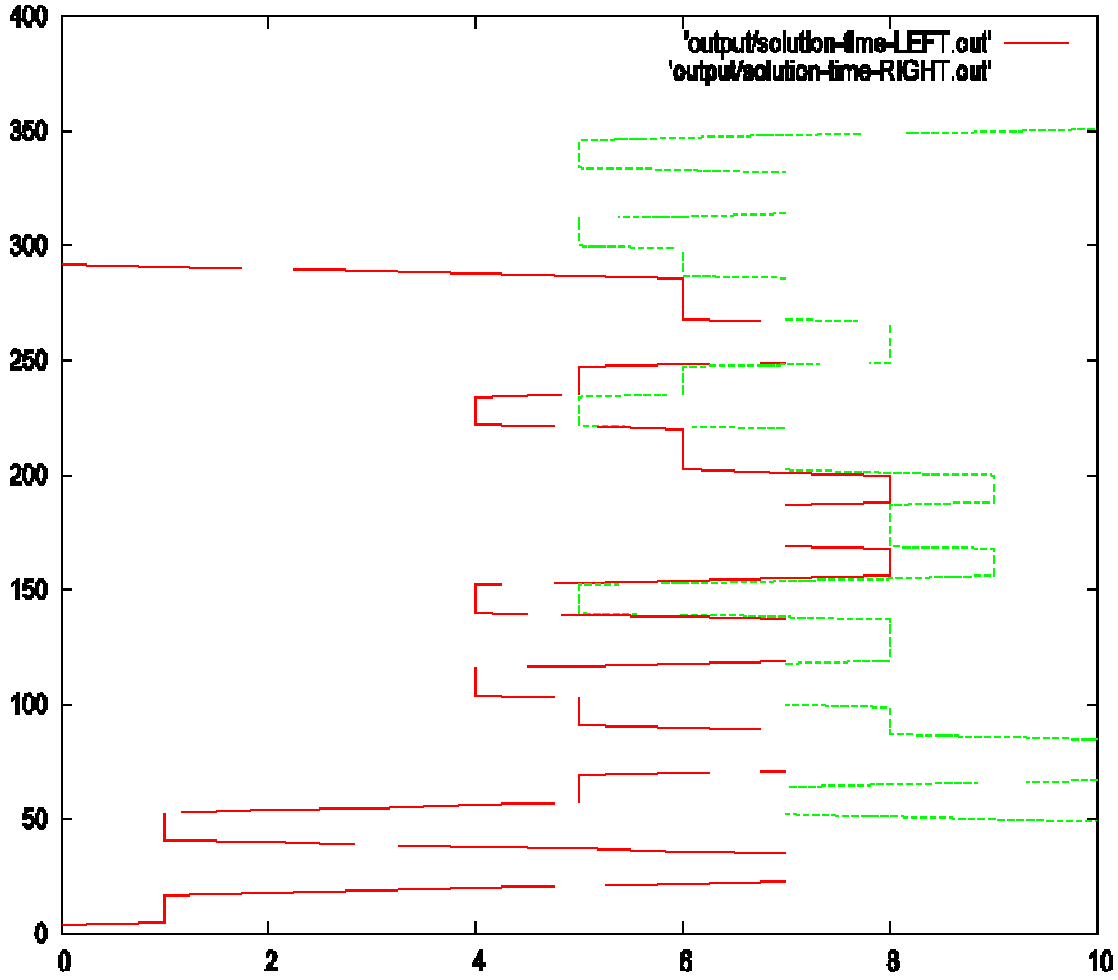
Computational Results

Problem 1: 60 jobs
Fixed-width time windows.

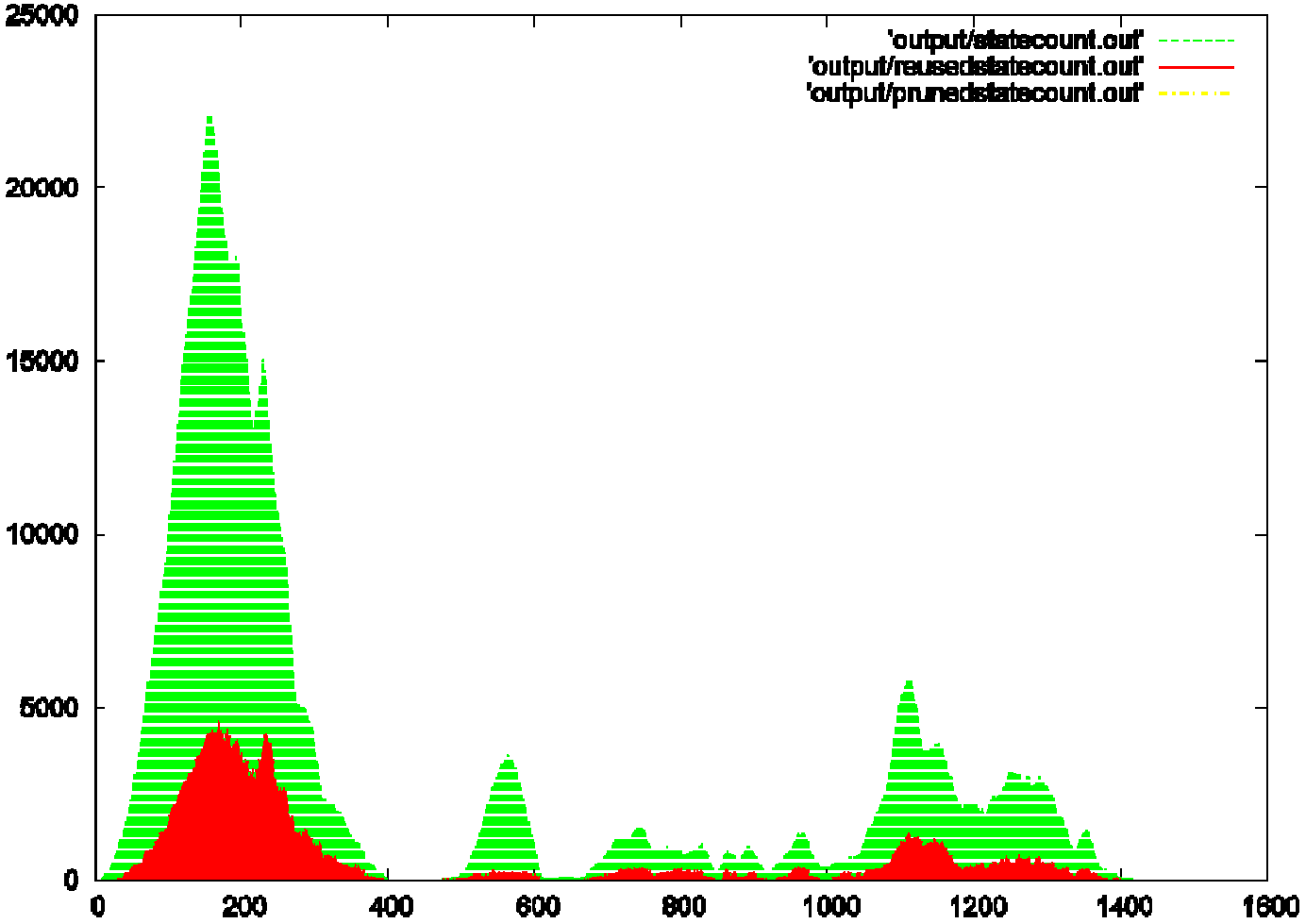
Jobs	Time window (mins)	Avg # states (optimal DP)	Peak # states (optimal DP)	Time for 10 rounds (secs)
10	25	3224	9477	158
20	35	3200	22,204	826
30	35	3204	22,204	1438*

*Multiple DPs solved in some rounds.

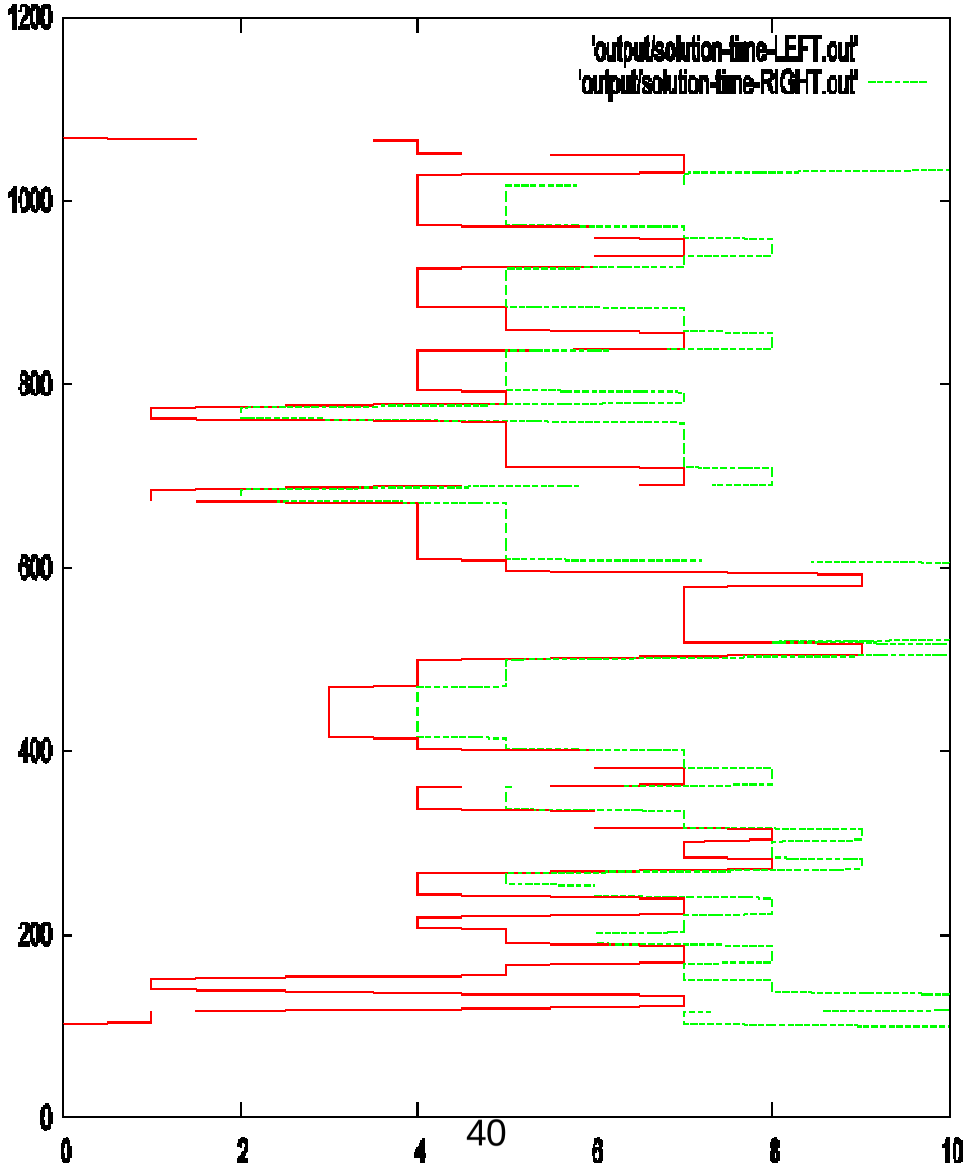
Solution of 10-job problem



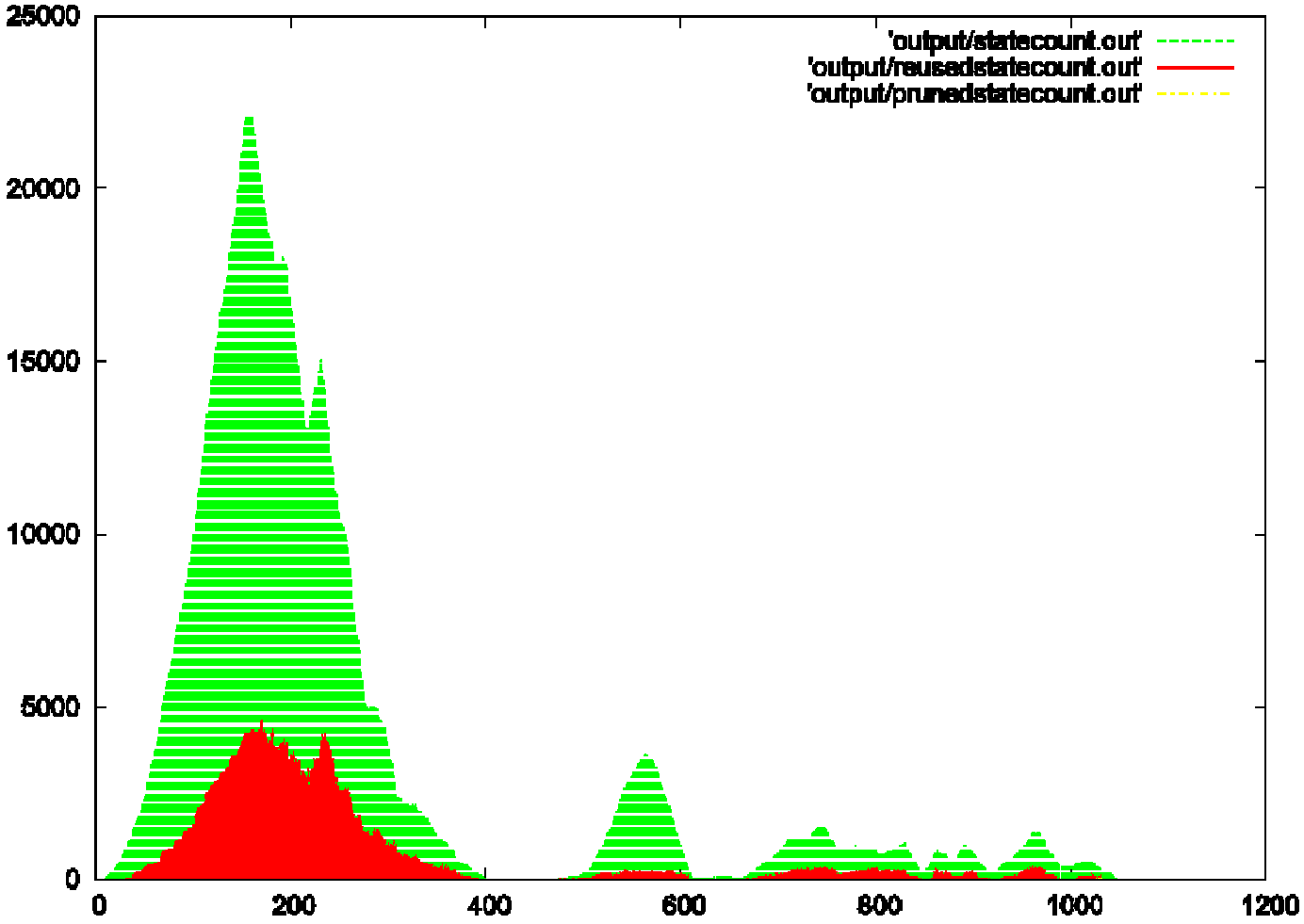
State space size for 10-job problem, for each time



Solution of 20-job problem



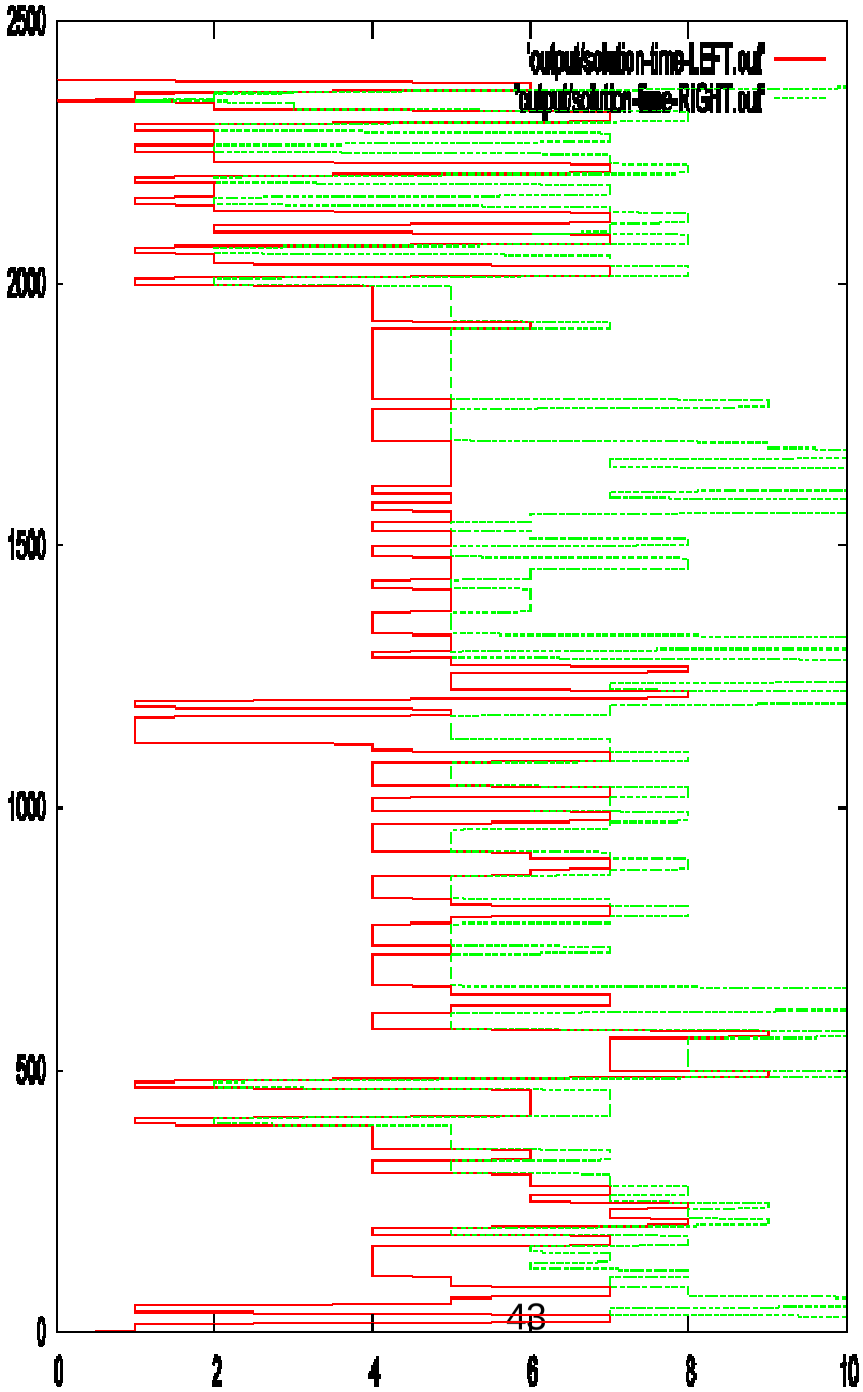
State space size for 20-job problem



Computational Results

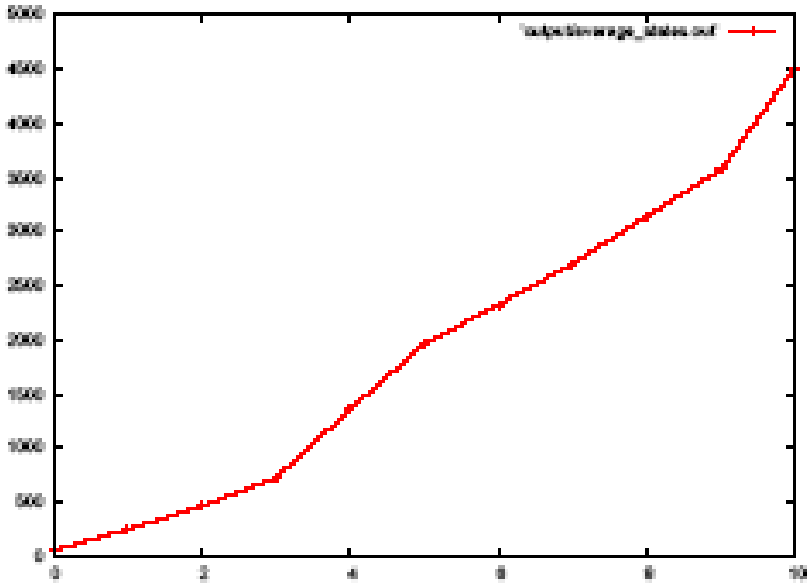
Problem 1: 60 jobs
Start with minimum time windows that allow feasibility.
Width varies from 5 to 104 minutes.
Add 5, 10 minutes

Jobs	Avg time window (mins)	Avg # states (optimal DP)	Peak # states (optimal DP)	Time for 10 rounds (seconds)
60	37	53	632	4
60	42	1948	9449	265
60	47	4492	20,232	1050

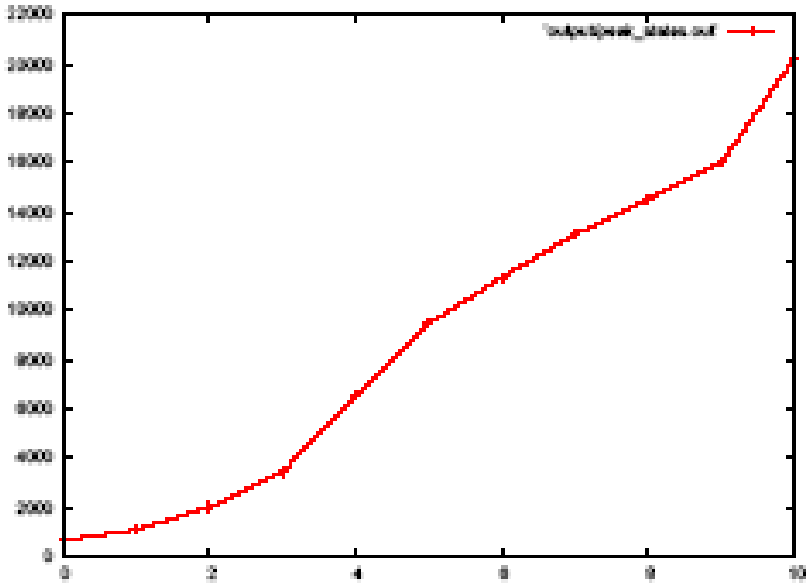


Solution of
60-job problem

State space size vs. minutes added to windows.

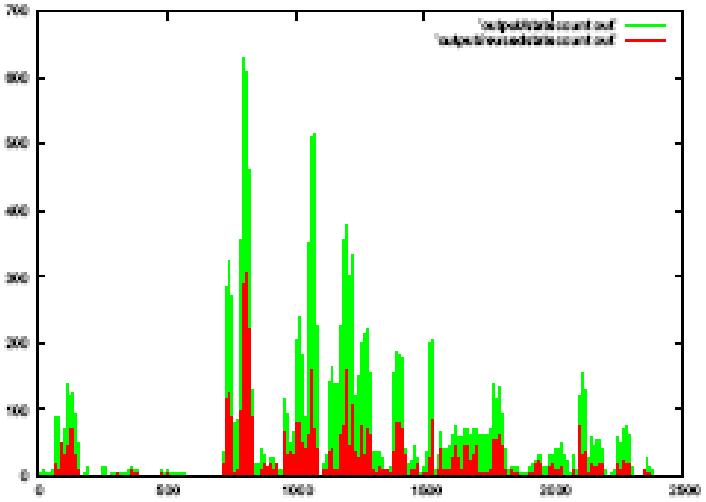


Average

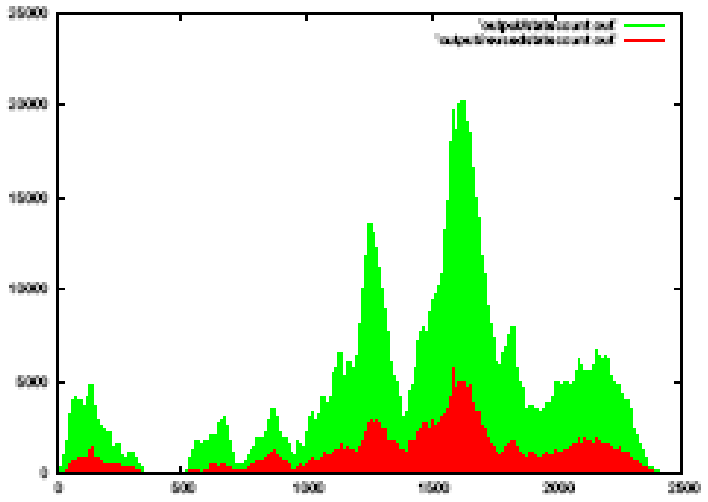


Max

State space size vs. time period



Minimum windows



Min windows + 10 mins

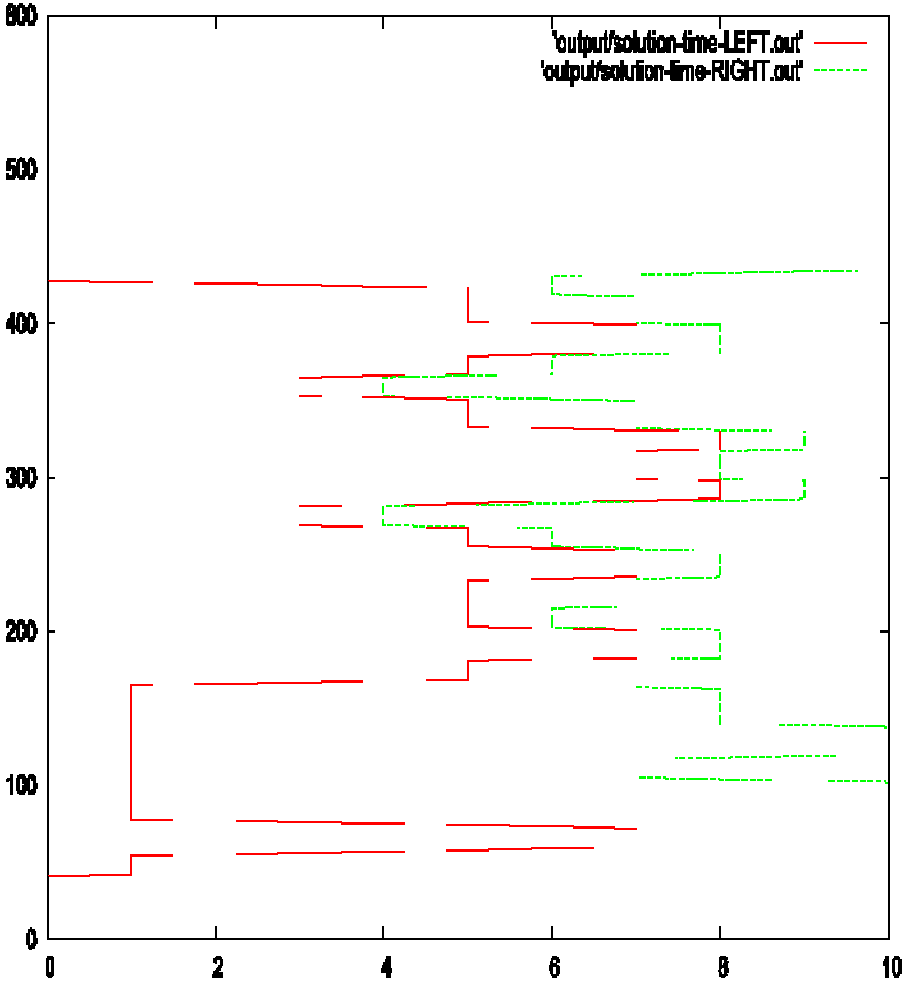
Computational Results

Problem 2: 258 jobs
Time windows specified by client.

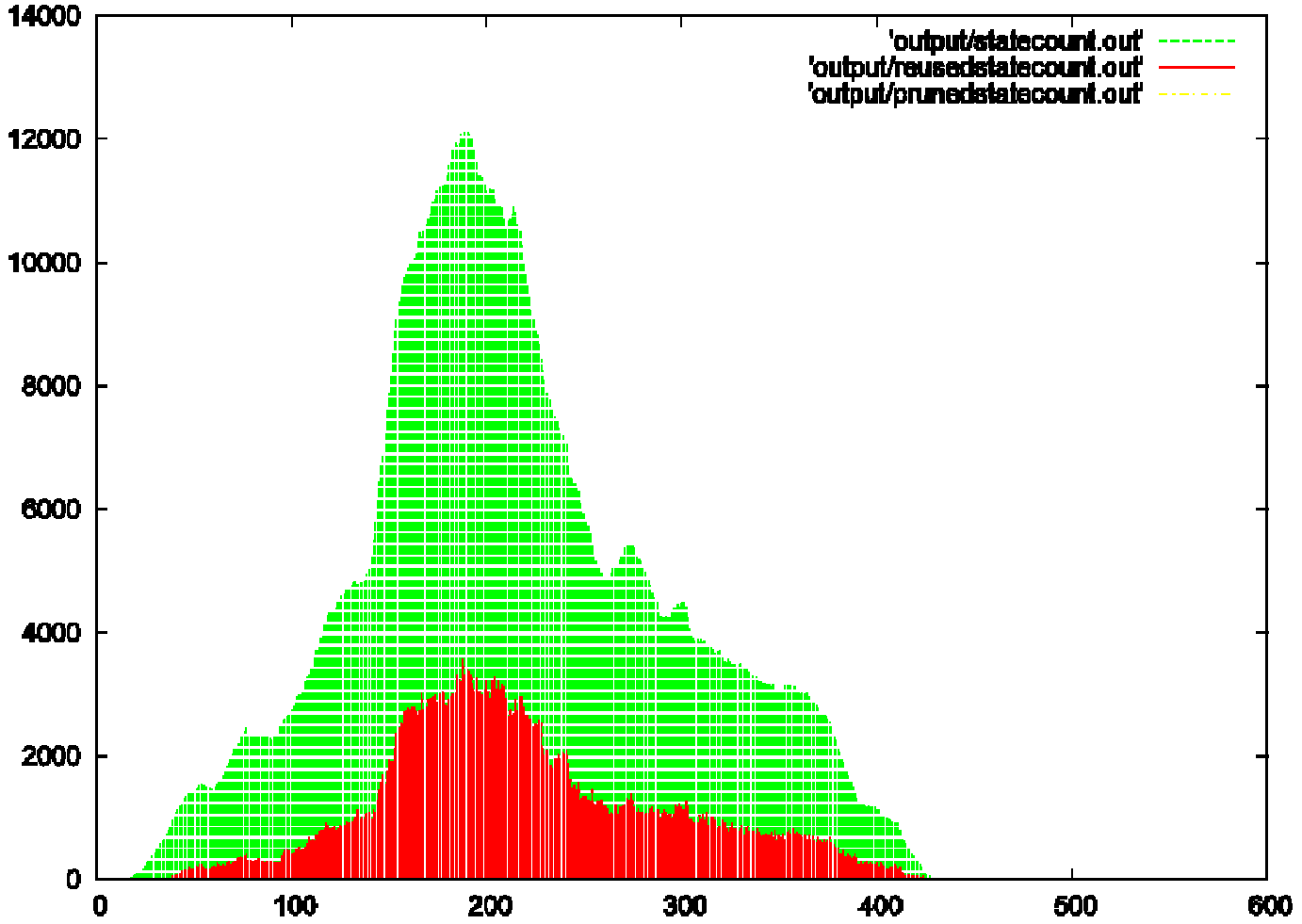
Jobs	Minimum time window padding to get feasible solution (mins)	Time for one DP (secs)
10	0	18
20	5	32*
30	30+	?

*Multiple DPs solved in some rounds.

Solution of 10-job problem



State space size for 10-job problem



Computational Results

Problem 2: 258 jobs
Adjusted release times.
Same time windows as before.

Jobs	Minimum time window padding to get feasible solution (mins)	Time for one DP (secs)
10	0	18
20	5	21
30	30+	?

*Multiple DPs solved in some rounds.

Conclusions

- It appears that, in many cases, cranes **must** lag behind a reasonable production schedule.
  - Exact algorithm finds no feasible trajectory.
- But it is not enough to spread out the release times.
  - Certain windows must be very large.
  - It is hard to predict which ones.
    - Due to combinatorial nature of problem.
  - So **all windows** must be stretched.

Conclusions

- When all windows are wide, the state space blows up.
- For speed, must solve problem sequentially, a few jobs at a time (e.g., 10).

What Next?

- Speed up the DP.
  - Exploit partial separability of state variables.
  - Still an exact algorithm, but may not be fast enough.
- Sacrifice exactness for speed.
  - Use a heuristic dispatching rule.
  - Rely on same theorem to shape trajectories.
  - May fail to find feasible solutions, but fast enough for many assignment/sequencing iterations.

Speed Up the DP

- Avoid enumerating states that are identical except for processing times.
  - Cranes are processing most of the time.
  - State variables are *location* and *task*.
  - Results in many fewer states.
- Compromise the Markovian property.
  - When a state has both cranes at stop locations for their current task…
    - store costs for different processing times in an array that persists for several time periods.

Speed up the DP

Compute these costs when a task pair for which both tasks are at their stop locations appears in the state space.

c_{41}				
c_{31}				
c_{21}				
c_{11}	c_{12}	c_{13}	c_{14}	c_{15}

c_{ij} = cost-to-go when the left crane has been processing *i* time units and the right crane has been processing *j* time units.

Speed Up the DP

Compute these costs in the next period. No need to update existing costs.

Data structure is a 2-dimensional circular queue.

c_{12}	c_{13}	c_{14}	c_{15}	c_{11}
c_{42}				c_{41}
c_{32}				c_{31}
c_{22}	c_{23}	c_{24}	c_{25}	c_{21}

c_{ij} = cost-to-go when the left crane has been processing i time units and the right crane has been processing j time units.

Speed Up the DP

And similarly in the next period.

c_{23}	c_{24}	c_{25}	c_{21}	c_{22}
c_{13}	c_{14}	c_{15}	c_{11}	c_{12}
c_{43}			c_{41}	c_{42}
c_{33}	c_{34}	c_{35}	c_{31}	c_{32}

c_{ij} = cost-to-go when the left crane has been processing *i* time units and the right crane has been processing *j* time units.

Speed Up the DP

These costs do not correspond to separate states.

After this point the table is no longer needed and memory can be released.

c_{34}	c_{35}	c_{31}	c_{32}	c_{33}
c_{24}	c_{25}	c_{21}	c_{22}	c_{23}
c_{14}	c_{15}	c_{11}	c_{12}	c_{13}
c_{44}	c_{45}	c_{41}	c_{42}	c_{43}

c_{ij} = cost-to-go when the left crane has been processing *i* time units and the right crane has been processing *j* time units.

Heuristic Dispatching

- Rely on same theorem used by DP to shape the trajectory.
  - Resulting trajectories are simpler and easier to implement.
- Use time windows differently.
  - Time window refers to lapse between start of first task in a job to completion of last task.
    - This is impractical in DP because it requires a state variable.
  - Production schedule can be revised to reflect resulting job start times.

Heuristic Dispatching

- Yield to the other crane only if time still remains to complete remaining tasks in the job.
  - Allow limited backtracking.
  - Dispatching rule fails if time runs out.
- Compare with DP on small problems.