Solving Exponential-Size Integer Programs Using Branch-and-Cut-and-Price

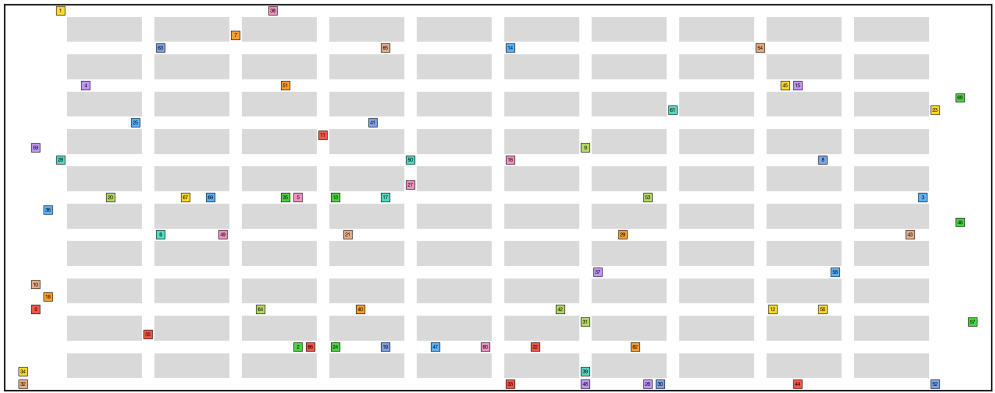
Edward Lam
edward.lam@monash.edu
ed-lam.com

Overview

- Multi-agent path finding problem
- Single-agent path finding problem
- Integer programming
- Exponential-size models
- Column generation
- Row generation
- Branching

Multi-Agent Path Finding

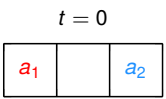
Multi-Agent Path Finding

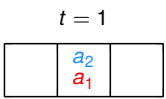


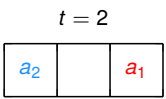
Multi-Agent Path Finding

- Given:
  - A grid environment
  - A set of agents
    - A start and goal location for each agent
- Aim:
  - Navigate every agent from its start to its goal
  - At any timestep, agents can move north, south, west, east or wait at its current location
  - Agents must avoid collisions with other agents (vertex conflicts and edge conflicts)
  - Minimize sum of arrival time
  - Agents can visit its goal location but not complete/arrive until a later revisit
  - Agents wait at its goal location upon arrival, blocking it indefinitely

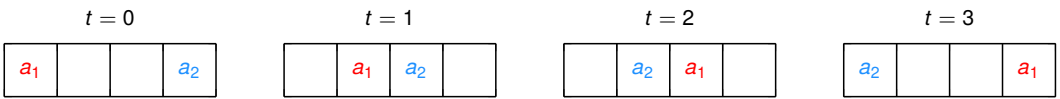
Vertex Conflicts







Edge Conflicts



Simplified Problem for Today

- Has vertex conflicts but no edge conflicts
- Agents disappear upon reaching its goal location, does not block indefinitely

Summary

- Defined the multi-agent path finding problem
  - Find a path for every agent from its start location to its goal location
  - Avoid vertex conflicts and edge conflicts
  - Minimize sum of arrival times
  - Block goal location indefinitely upon arrival
- Defined simplified variant
  - Vertex conflicts only, no edge conflicts
  - Agents disappear upon completion

Single-Agent Path Finding

Shortest Path Problem

- Given:
  - A graph $G = (V, E)$ with vertices V and edges E
  - A *start vertex* $v_{\text{start}} \in V$
  - A set of *goal vertices* $V_{\text{goal}} \subseteq V$
  - A cost function $c : E \to \mathbb{R}_+$
    - Every edge $(u, v) \in E$ has cost $c(u, v) \geq 0$
- A *path* or *solution* $p = (v_0, \ldots, v_{n-1})$ of *length* n is a sequence of n vertices such that $v_0 = v_{\text{start}}$, $v_{n-1} \in V_{\text{goal}}$ and $(v_t, v_{t+1}) \in E$ for all $t \in \{0, \ldots, n-2\}$
- The cost of p is

$$c(p) = \sum_{t=0}^{n-2} c(v_t, v_{t+1})$$

- The *shortest path problem* asks to find a path with minimum cost, i.e.

$$\underset{p \text{ is a path}}{\arg\min}\, c(p)$$

Dijkstra's Algorithm

- Well-known optimal algorithm[1]
- Maintain a priority queue representing a search frontier called the open set
- Searches outwards from the starting vertex in waves
- The $g(v)$ value of a vertex v is the lowest known cost from v_{start} to $v \in V$

[1] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik, 1:269-271, 1959.

Dijkstra's Algorithm

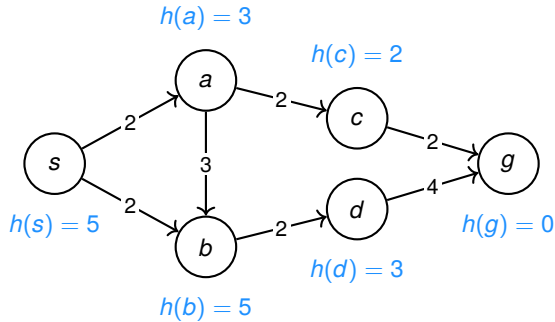
```
1  Function Dijkstra(G = (V, E), v_start, V_goal, c):
2      for v ∈ V do
3          g(v) ← ∞
4          parent(v) ← NULL
5      g(v_start) ← 0
6      OPEN ← {v_start}
7      while |OPEN| > 0 do
8          u ← arg min_{u∈OPEN} g(u)
9          if u ∈ V_goal then
10             return ReconstructPath(u, parent)
11         OPEN ← OPEN \ {u}
12         for (u, v) ∈ E do
13             if g(u) + c(u, v) < g(v) then
14                 g(v) ← g(u) + c(u, v)
15                 parent(v) ← u
16                 OPEN ← OPEN ∪ {v}
17     return NULL
```

Demo

https://www.movingai.com/SAS/ASM/

Heuristic Functions

- A *heuristic function*[2] $h(v)$ approximates the cost to-go from $v \in V$ to every $v_{\text{goal}} \in V_{\text{goal}}$
- The term "heuristic" is just the name of h, and does not refer to any notion of suboptimality
- A heuristic is *admissible* if it never over-estimates the cost to-go
- An admissible heuristic is called a *completion bound* in the column generation literature



[2] S. Russell and P. Norvig, Artificial intelligence: a modern approach, Pearson, 4th edition, 2020.

A* Algorithm

- Classical AI planning algorithm[3]
- Still the workhorse algorithm in use today
- A* requires a heuristic function to direct the search towards a goal
- A* is optimal if the heuristic is admissible
  - If all edges have non-negative cost, then $h(v) = 0$ for all $v \in V$ is admissible
- Assume admissibility from now on
- The priority queue is ordered by $f(v) := g(v) + h(v)$ instead of only $g(v)$ as in Dijkstra

[3] P.E. Hart, N.J. Nilsson, B. Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Transactions on Systems Science and Cybernetics, 4(2):100-7, 1968.

A* Algorithm

```
1  Function A*(G = (V, E), v_start, V_goal, c, h):
2      for v ∈ V do
3          g(v) ← ∞
4          parent(v) ← NULL
5      g(v_start) ← 0
6      OPEN ← {v_start}
7      while |OPEN| > 0 do
8          u ← arg min_{u∈OPEN} f(u)
9          if u ∈ V_goal then
10             return ReconstructPath(u, parent)
11         OPEN ← OPEN \ {u}
12         for (u, v) ∈ E do
13             if g(u) + c(u, v) < g(v) then
14                 g(v) ← g(u) + c(u, v)
15                 parent(v) ← u
16                 OPEN ← OPEN ∪ {v}
17     return NULL
```

Demo

https://www.movingai.com/SAS/ASM/

Single-Agent Path Finding

- Given:
  - A set of 2D locations $L = \{(x_1, y_1), \ldots, (x_{|L|}, y_{|L|})\}$
  - An infinite set of timesteps $T = \{0, 1, \ldots\}$
  - A start location $l_{\text{start}} \in L$ and goal location $l_{\text{goal}} = (x_{\text{goal}}, y_{\text{goal}}) \in L$, possibly with $l_{\text{start}} = l_{\text{goal}}$
- Define a time-expanded graph $G = (V, E)$ where $V = L \times T$ and $E = \{(((x_1, y_1), t_1), ((x_2, y_2), t_2)) \in V \times V : |x_1 - x_2| + |y_1 - y_2| \leq 1 \wedge t_2 = t_1 + 1\}$
  - The edges represent the agent moving north, south, west, east or waiting at its current location from one timestep to the next
- *Manhattan distance* heuristic $h((x, y), t) = |x - x_{\text{goal}}| + |y - y_{\text{goal}}|$
- Every edge has cost 1 (including wait actions)
- Find a shortest path from $(l_{\text{start}}, 0)$ to any goal in $V_{\text{goal}} = \{(l, t) \in V : l = l_{\text{goal}} \wedge t \in T\}$

Summary

- Defined a shortest path problem
- Defined a heuristic function, admissibility
- Described Dijkstra's algorithm and A* for optimal shortest path
- Defined the single-agent path finding problem

Task 1

1. Download the template code
   - `git clone https://github.com/ed-lam/cpaior2025-master-class`
2. Install PySCIPOpt, the Python interface for SCIP
   - `pip install pyscipopt`
3. Find `[TASK1]` in `pricer.py`
   - Implement the `edge_cost` function which computes the cost of an edge
   - Hint: every edge has cost `1.0`
4. Run your work
   - `python task1_astar.py instances/empty-16-16-random-9.scen`

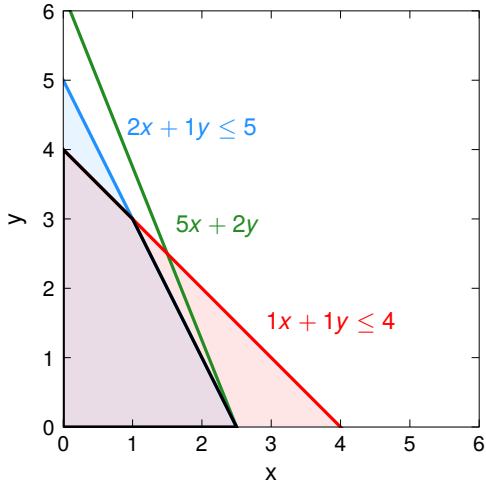
Integer Programming

Linear Programs

- A *linear program*, *linear programming model* or *linear programming formulation* consist of:
  - *Variables*: decisions taking real values
  - *Constraints*: linear equations or inequalities stating relationships between the variables
  - *Objective function*: evaluate the quality of a solution
  - *Optimisation direction*: minimize or maximize the objective function
    - Trivial to transform one to other

Example of a Linear Program

$$
\begin{aligned}
\text{max} \quad & 5x + 2y \\
\text{subject to} \quad & 2x + 1y \leq 5 \\
& 1x + 1y \leq 4 \\
& x \geq 0 \\
& y \geq 0
\end{aligned}
$$

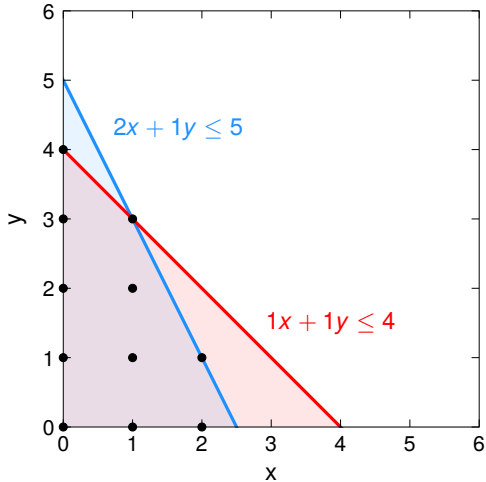


Integer Programs

- An *integer program* is a linear program where all variables must take integer values instead of real values

Example of an Integer Program

$$
\begin{aligned}
\max \quad & 5x + 2y \\
\text{subject to } & 2x + 1y \leq 5 \\
& 1x + 1y \leq 4 \\
& x \geq 0 \\
& y \geq 0 \\
& x, \quad y \text{ integer}
\end{aligned}
$$

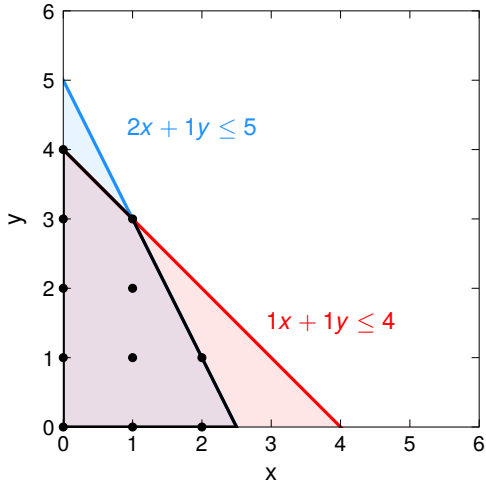


Solving Integer Programs

- Declare a model
- Input the model into a black-box solver (e.g., Gurobi, CPLEX)
- Wait for a solution
  - Could be a long time for large instances

Branch-and-Bound Algorithm

- Most solvers implement a *branch-and-bound* tree search
- At every node of the tree, solve a *continuous relaxation* (e.g., linear)
  - A linear relaxation is a *linear program* that ignores the *integrality constraints*
- If the solution is fractional, *branch*
  - Create children nodes that do not contain the current solution
  - e.g., $\hat{x} = 2.5$, then impose $x \leq 2$ in "down" child and $x \geq 3$ in "up" child



Summary

- Defined a linear program and integer program
- Briefly introduced the branch-and-bound algorithm for solving integer programs
- Defined the linear relaxation of an integer program

Exponential-Size Models

Model Size

- Let n be the input size
  - Think of it as the number of tasks, items, nodes, etc.
- A model is *polynomial-size* if the growth of the encoding (variables, constraints and coefficients) is bounded by a polynomial function in n
- Otherwise, it is *exponential-size*

Traveling Salesman Example

- Minimize distance to visit n cities, each exactly once, then return to the starting city
- Two common families of constraints for subtour elimination
  - Polynomially-many Miller-Tucker-Zemlin (MTZ) constraints[4]

$$u_i - u_j + nx_{i,j} \leq n - 1, \quad \forall i \in \{2, \ldots, n\}, j \in \{2, \ldots, n\}, i \neq j$$

  - Exponentially-many Dantzig-Fulkerson-Johnson (DFJ) constraints[5]

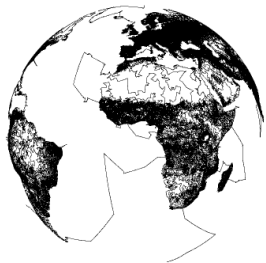
$$\sum_{i \in S} \sum_{j \in S, j \neq i} x_{i,j} \leq |S| - 1, \quad \forall S \subseteq \{1, \ldots, n\}, 2 \leq |S| \leq n - 1$$

[4] C.E. Miller, A.W. Tucker, R.A. Zemlin, Integer programming formulation of traveling salesman problems Journal of the ACM, 7(4):326-329, 1960.

[5] G. B. Dantzig, R. Fulkerson, S. Johnson, Solution of a large-scale traveling-salesman problem, Journal of the Operations Research Society of America, 2(4):393-410, 1954.

Traveling Salesman Example

- The Concorde TSP solver closed all TSPLIB instances using an exponential-size model (up to 85,900 cities)[6]
- Concorde computed the best known lower bound to the World TSP instance (1.9 million cities)



[6] D.L. Applegate, R.E. Bixby, V. Chvatál, W.J. Cook, The traveling salesman problem: a computational study, Princeton University Press, 2006.

Bin Packing Example

- Given m bins, each with a common capacity, and n items, each with some demand
- Minimize number of bins required to store all items without exceeding capacity
- Polynomial-size model: decide which of m bins that every item is put in
- Exponential-size model: enumerate all possible patterns and choose m patterns

MAPF Example

- Polynomial-size model: decide on an agent moving north, south, east, west or waiting at every timestep
- Exponential-size model: enumerate all possible paths for every agent and choose one path

Pros and Cons of Exponential-Size Models

- Downsides
  - Huge number of variables and constraints
- Benefits
  - Stronger lower bound
  - Eliminate symmetry
  - Exploit substructure
  - Richer geometry

Stronger Lower Bound

- Lower bound is no worse, often much stronger than the equivalent compact model, in theory and in practice[7]
- Stronger lower bound = smaller search tree
- Can shrink the tree from 100,000s nodes to 100s nodes
- No free lunch: NP-hard problem, so have to pay for computation somewhere

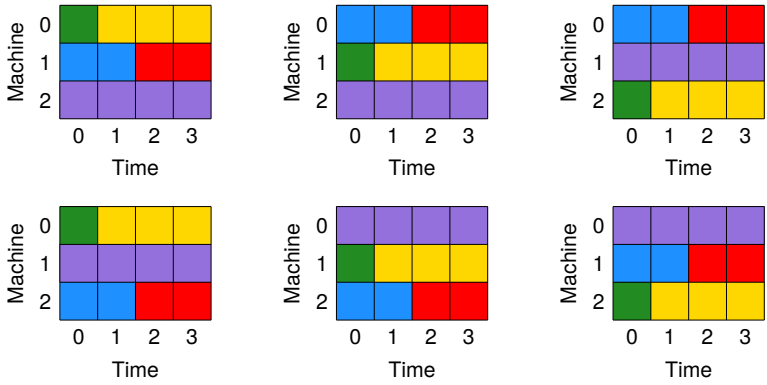
[7] M.E. Lübbecke, J. Desrosiers, Selected topics in column generation, Operations Research, 53(6):1007-1023, 2005.

Eliminate Symmetry

- Remove index of identical machines, vehicles, actors, agents, etc.
- Formulate a model to select n plans instead of assigning tasks to n machines
- No need to break symmetry (breaking symmetry can be slower[8])

[8] G. Chu, P.J. Stuckey, Dominance Driven Search, CP 2013.

Parallel Machine Scheduling Example

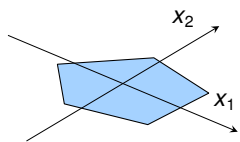


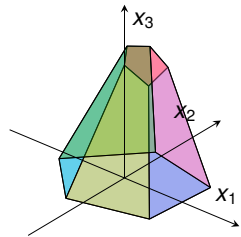
Exploit Substructure

- Hide the decisions/actions of each machine inside the definition of a “plan” or “pattern”
- Use specialised algorithms to generate plans
  - Often want plans to correspond to paths
  - Therefore use shortest path algorithms

Richer Geometry

- Every variable corresponds to a dimension
- Work in higher dimensional space and its projection on to a lower dimensional space
- Define “robust” constraints in lower dimensions and “non-robust” constraints in the higher dimensions
  - Subset row[9] for vehicle routing and target conflicts[10] for MAPF





[9] M. Jepsen, B. Petersen, S. Spoorendonk, D. Pisinger, Subset-row inequalities applied to the vehicle-routing problem with time windows, Operations Research, 56(2):497-511, 2008.

[10] E. Lam, P. Le Bodic, D. Harabor, and P.J. Stuckey, Branch-and-cut-and-price for multi-agent path finding, Computers & Operations Research, 144:105809, 2022.

Solving Polynomial-Size Models

- Polynomial-size models can usually be entered into a blackbox solver (e.g., Gurobi, CPLEX, OR-Tools, SCIP, HiGHS, etc.)

Solving Exponential-Size Models

- Exponential-size models are too big to construct in a computer
- Generate a subset of variables and constraints on-the-fly
  - Generate variables using *column generation* algorithm
  - Generate constraints using *cutting plane* algorithm
- The final subset is much smaller than full set but still much larger than polynomial-size models
- Sometimes faster to solve exponential-size models
  - State-of-the-art for vehicle routing[11] and multi-agent path finding[12]

[11] L. Costa, C. Contardo, G. Desaulniers, Exact branch-price-and-cut algorithms for vehicle routing, Transportation Science, 53(4):946-985, 2019

[12] E. Lam and P.J. Stuckey, Low-level search on time intervals in branch-and-cut-and-price for multi-agent path finding, SOCS2025.

Summary

- Described the size of integer programming models
- Highlighted the benefits and drawbacks of exponential-size models
  - Stronger lower bound
  - Eliminate symmetry
  - Exploit substructure
  - Richer geometry
  - Huge number of variables and constraints
- Named two algorithms for dynamically generating variables and constraints

Column Generation

Column Generation

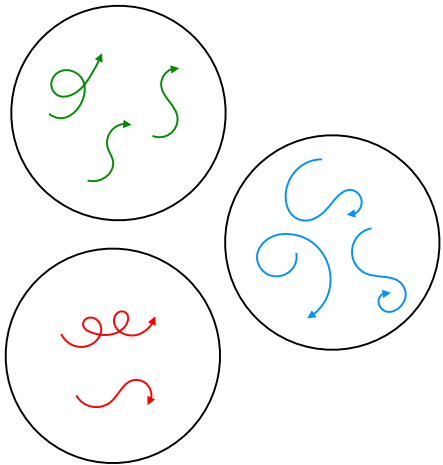
- *Column generation* is an algorithm for solving linear programs with large number of variables
  - Generate a subset of the variables on demand

Two-Level Model

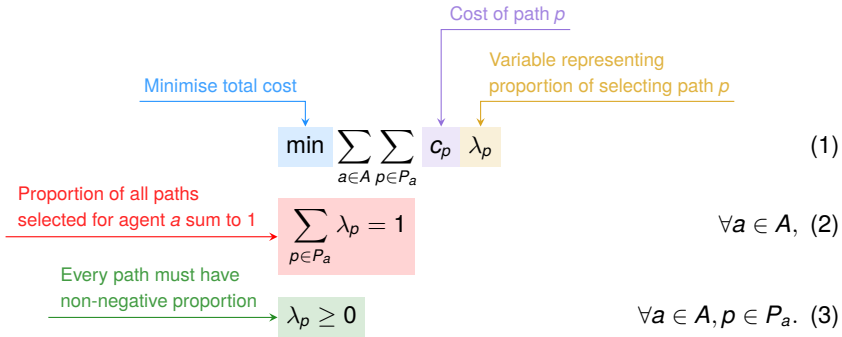
- Low-level *pricing problem*: generate individual plans
  - Generate plans for each staff member/vehicle/agent
  - Generate plans for each type of identical machine
- High-level *master problem*: assemble the known plans together subject to overarching constraints
  - Choose 1 plan for each staff member/vehicle/agent
  - Choose n_i plans for machines of type i

MAPF Master Problem

- Let P_a be the set of paths for every agent $a \in A$
- Variable $\lambda_p \in [0, 1]$ be the proportion of selecting path $p \in P_a$
- Path selection constraints: for every agent a, select paths from P_a such that the total proportion sums to 1
- Objective: minimise cost of paths selected



MAPF Master Problem



MAPF Example

p	1	2	3	4	5		
c_p	9	9	2	7	5	Sign	RHS
Agent 1	1	1	0	0	0	$=$	1
Agent 2	0	0	1	0	0	$=$	1
Agent 3	0	0	0	1	1	$=$	1
$\hat{\lambda}_p$	0.5	0.5	1	0	1		

Cost of solution $\hat{\lambda} = 9 \times 0.5 + 9 \times 0.5 + 2 \times 1 + 7 \times 0 + 5 \times 1 = 16$

Reduced Cost

- After solving the master problem, retrieve a *dual multiplier* $\bar{\pi}_i$ for every row i

p	1	2	3	4	5			
c_p	9	9	2	7	5	Sign	RHS	$\bar{\pi}_i$
Agent 1	1	1	0	0	0	$=$	1	9
Agent 2	0	0	1	0	0	$=$	1	2
Agent 3	0	0	0	1	1	$=$	1	5
$\hat{\lambda}_p$	0.5	0.5	1	0	1			

- The *reduced cost* $\bar{c}_p$ of any plan p is

$$\bar{c}_p := c_p - \sum_{\text{rows } i} (\text{entry in row } i) \times \bar{\pi}_i$$

Improving Columns

p	1	2	3	4	5			
c_p	9	9	2	7	5	Sign	RHS	$\bar{\pi}_i$
Agent 1	1	1	0	0	0	$=$	1	9
Agent 2	0	0	1	0	0	$=$	1	2
Agent 3	0	0	0	1	1	$=$	1	5

- Say we know a plan $p = 6$ for agent 3 with $c_6 = 3$, then

$$\bar{c}_p := c_p - \sum_{\text{rows } i} (\text{entry in row } i) \times \bar{\pi}_i = 3 - 1 \times 5 = -2$$

- Since $\bar{c}_6 < 0$, inserting plan 6 as a new column will obtain a better[13] solution
- Then solve the master problem again

[13] Actually, no worse due to degeneracy

MAPF Example

Before

p	1	2	3	4	5		
c_p	9	9	2	7	5	Sign	RHS
Agent 1	1	1	0	0	0	$=$	1
Agent 2	0	0	1	0	0	$=$	1
Agent 3	0	0	0	1	1	$=$	1
$\hat{\lambda}_p$	0.5	0.5	1	0	1		

Cost = $9 \times 0.5 + 9 \times 0.5 + 2 \times 1 +$
$7 \times 0 + 5 \times 1 = 16$

After

p	1	2	3	4	5	6		
c_p	9	9	2	7	5	3	Sign	RHS
Agent 1	1	1	0	0	0	0	$=$	1
Agent 2	0	0	1	0	0	0	$=$	1
Agent 3	0	0	0	1	1	1	$=$	1
$\hat{\lambda}_p$	0.5	0.5	1	0	0	1		

Cost = $9 \times 0.5 + 9 \times 0.5 + 2 \times 1 +$
$7 \times 0 + 5 \times 0 + 3 \times 1 = 14$

Reduced Cost

- The reduced cost of a column/plan represents the maximum change in the cost of the master problem per unit increase from its current value
- Plans that do not yet exist have implicit value of 0
- There may be other constraints that prevent it from attaining this maximum change

Pricing Problem

- The *reduced cost* $\bar{c}_p$ of any plan p is

$$\bar{c}_p := c_p - \sum_{\text{rows } i} (\text{entry in row } i) \times \bar{\pi}_i$$

- The *pricing problem* for any agent $a \in A$ asks to find a path

$$p^* = \arg\min_{p \in P_a} \bar{c}_p$$

  with $\bar{c}_{p^*} < 0$
- Design your model carefully so that P_a represents a problem for which we have specialised algorithms
- An algorithm for solving the pricing problem is called a *pricing algorithm* or a *pricer*

Column Generation Loop

1. Solve the master problem to obtain a (primal) solution $\hat{\lambda}$ and dual multipliers (solution) $\bar{\pi}$
2. Using $\bar{\pi}$, solve the pricing problem to find columns with minimum reduced cost
3. If the minimum is negative, add the corresponding columns to the master problem and go to Step 1
4. Otherwise, terminate because $\hat{\lambda}$ is optimal (for the linear relaxation)

Initialisation

- The master problem could be infeasible because there are no columns (initially at the root node or after branching)
- Initialise with columns corresponding to a solution from another algorithm
- Run pricer with *Farkas cost*[14] function instead of *reduced cost*
- The Farkas cost $\tilde{c}_p$ of any plan p is

$$\tilde{c}_p := -\sum_{\text{rows } i} (\text{entry in row } i) \times \tilde{\pi}_i$$

  where $\tilde{\pi}$ are Farkas multipliers
- Compare to the reduced cost

$$\bar{c}_p := c_p - \sum_{\text{rows } i} (\text{entry in row } i) \times \bar{\pi}_i$$

- Note some solvers (Gurobi) will report the wrong sign so multiply by -1

[14] E.D. Andersen, How to use Farkas' lemma to say something important about infeasible linear problems, technical report, MOSEK, 2014.

Summary

- Use column generation to generate only a subset of all variables
  - Solve a pricing problem to find plans with negative reduced cost
  - Add these plans as a column and iterate until none exist
- If the master problem is infeasible, restore feasibility by pricing with Farkas cost function

Task 2

1. Find `[TASK2]` in `pricer.py`
   - The reduced cost of path $p = (v_0, \ldots, v_{n-1})$ for any agent $a \in A$ is

$$\bar{c}_p = c_p - \bar{\pi}_a = \sum_{t=0}^{n-2} c(v_t, v_{t+1}) - \bar{\pi}_a$$

Original cost

Dual multiplier for agent constraint of agent a

Cost of edge from v_t to v_{t+1}

   - The Farkas cost of path p for any agent $a \in A$ is

$$\tilde{c}_p = -\tilde{\pi}_a$$

Farkas multiplier

   - Retrieve the dual multiplier (either $\bar{\pi}_a$ or $\tilde{\pi}_a$) of the agent constraint in the master problem
   - Implement the offset $-\bar{\pi}_a$ or $-\tilde{\pi}_a$ in the cost calculation in the A* code
2. Run your work
   - `python task2_bcp.py --num-agents 55 instances/empty-16-16-random-9.scen`

Row Generation

Separator

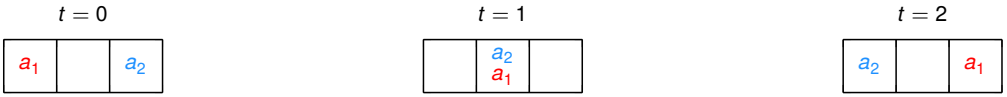
- A *row*, *constraint* or *cut* are synonyms
- A *separator* for one family of constraints is an algorithm that checks if a solution violates a constraint and adds the constraint to the master problem

MAPF Vertex Conflicts

- A vertex $v = (l, t)$ is a pair of a location and timestep
  - Infinite number of vertices
- Many vertices will never be in conflict because agents never go there
- So ignore the vertex conflict constraints initially and generate them on-demand

MAPF Vertex Conflicts

Step 1: Initial solve selecting lowest cost paths.



Step 2: Call the separator for vertex conflict conditions to add a vertex conflict constraint.

Step 3: Solve again.

Separator for Vertex Conflicts

- A vertex conflict occurs whenever a vertex $v = (l, t)$ is used with proportion more than 1
- Define $x_v^p = 1$ if path p uses vertex v, and $x_v^p = 0$ otherwise
- Calculate the proportion x_v of using a vertex v from the path selections

$$x_v = \sum_{a \in A} \sum_{p \in P_a} x_v^p \lambda_p$$

- If $x_v > 1$, add the constraint

$$\sum_{a \in A} \sum_{p \in P_a} x_v^p \lambda_p \leq 1$$

MAPF Example

Before

p	1	2	3	4	5	6		
c_p	9	9	2	7	5	3	Sign	RHS
Agent 1	1	1	0	0	0	0	$=$	1
Agent 2	0	0	1	0	0	0	$=$	1
Agent 3	0	0	0	1	1	1	$=$	1
$\hat{\lambda}_p$	0.5	0.5	1	0	0	1		

After

p	1	2	3	4	5	6		
c_p	9	9	2	7	5	3	Sign	RHS
Agent 1	1	1	0	0	0	0	$=$	1
Agent 2	0	0	1	0	0	0	$=$	1
Agent 3	0	0	0	1	1	1	$=$	1
Vertex conflict at (L, T)	0	0	1	0	0	1	$\leq$	1
$\hat{\lambda}_p$	0.5	0.5	1	0	1	0		

MAPF Example

- The pricing problem didn't know about this constraint (vertex conflict at (L, T))
- Generated a path that uses (L, T)
- The separator adds a constraint preventing (L, T) from being used with proportion > 1

Back to the Pricing Problem

- The pricer now sees the new dual multiplier

p	1	2	3	4	5	6			
c_p	9	9	2	7	5	3	Sign	RHS	$\bar{\pi}_i$
Agent 1	1	1	0	0	0	0	$=$	1	9
Agent 2	0	0	1	0	0	0	$=$	1	4
Agent 3	0	0	0	1	1	1	$=$	1	5
Vertex conflict at (L, T)	0	0	1	0	0	1	$\leq$	1	-2

- If any path p uses vertex (L, T), it will have coefficient 1 in the new row
- Reduced cost $\bar{c}_p$ of any plan p is

$$\bar{c}_p := c_p - \sum_{\text{rows } i} (\text{entry in row } i) \times \pi_i$$

- In the shortest path algorithm, if the candidate path visits (L, T), pay $-\pi_4$ in the reduced cost for contesting the resource

Sign of Dual Multipliers

When all coefficients are non-negative in a row and the master is a minimisation problem

Sign	Dual Multiplier
$\leq$	≤ 0
$=$	any
$\geq$	≥ 0

Price-and-Cut Loop

1. Solve the master problem to obtain a (primal) solution $\hat{\lambda}$ and dual multipliers (solution) $\bar{\pi}$
2. Using $\bar{\pi}$, solve the pricing problem to find columns with minimum reduced cost
3. If the minimum is negative, add the corresponding columns to the master problem and go to Step 1
4. Otherwise, run separator for every family of constraints
5. If violated constraints are found, add the corresponding rows to the master problem and go to Step 1
6. Otherwise, terminate because $\hat{\lambda}$ is optimal (for the linear relaxation)
    - The absent variables and constraints are not necessary

Summary

1. Defined a separator as a subroutine that checks for violated constraints and adds them to the master problem
2. Worked through vertex conflict constraints for MAPF
3. Showed the impact of adding a new constraint in the reduced cost function in the pricing problem
4. Outlined the price-and-cut loop for solving large linear programs

Task 3

1. `[TASK3a]` in `vertex_conflict_separator.py`
   - The code already determines which vertices are in conflict in the current solution
   - Using these vertices, create a constraint preventing every current vertex conflict from occurring
2. `[TASK3b]` in `pricer.py`
   - Store the negative dual multiplier $-\bar{\pi}_v$ of every vertex conflict v as a cost penalty inside the A*
3. `[TASK3c]` in `pricer.py`
   - The reduced cost of path $p = (v_0, \ldots, v_{n-1})$ for any agent $a \in A$ is

$$\bar{c}_p = c_p - \bar{\pi}_a - \sum_{\text{vertex conflict at } v} x_v^p \, \bar{\pi}_v$$

Indicates if p visits v

Dual multiplier for conflict constraint at v

   - Pay the penalty $-\bar{\pi}_v$ whenever the A* algorithm visits a vertex v in conflict
4. `[TASK3d]` in `master.py`
   - Read over this block of code which adds a coefficient to the column of incoming paths

Branching

MAPF Example

After adding cuts, you may still get a *fractional solution*

Branching Rules

- After adding cuts, you may still get a *fractional solution*, a feasible solution with fractional values
- Want an *integer solution*, corresponding to a solution to the problem
- A *branching rule* defines how to take a fractional solution at a node of the branch-and-bound tree and create children nodes that do not admit this fractional solution

Branching in Column Generation

- In standard integer programming, choose a variable x with fractional value $\hat{x}$ and impose $x \leq \lfloor \hat{x} \rfloor$ in one child and $x \leq \lceil \hat{x} \rceil$ in the other child
  - e.g., $\hat{x} = 2.5$, then impose $x \leq 2$ in "down" child and $x \geq 3$ in "up" child
  - *Most fractional branching* (i.e., choosing an x whose fractional part is closest to 0.5) is worse than random selection[15]
- Cannot branch like this on the λ variables in column generation
  - What does $\lambda_p \leq 0$ and $\lambda_p \geq 1$ mean for one particular path p?
  - Too hard to impose this dual multiplier one one particular path in the pricing problem
- Need to branch on the *implicit variables* from the compact (polynomial-size) model
  - e.g., if a λ variable represents a path, branch on the edges making up a path

[15] T. Achterberg, T. Koch, A. Martin, Branching rules revisited, Operations Research Letters, 33(1):42-54, 2005

Application-Specific Branching Rules

- Branching rules are usually application-specific in column generation
- General-purpose branching rules are difficult to implement[16]
- Common to branch on the number of machines first, then branch on edges (successors)
- For MAPF, branch on path length first, then branch on vertices[17]

[16] F. Vanderbeck, On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm, Operations Research, 48(1):111-128, 2000.

[17] E. Lam, P. Le Bodic, D. Harabor, and P.J. Stuckey, Branch-and-cut-and-price for multi-agent path finding, Computers & Operations Research, 144:105809, 2022.

Realising Branching Decisions

- After finding a decision to impose, need to actually realise the decision
- The decision can be realised by:
  1. Adding a constraint local to that node and its subtree
     - This local constraint will have a dual multiplier, treat it like normal
  2. Remove all existing paths incompatible with the decision in the master problem (or set their domain to 0), and prevent incompatible paths from being generated in the pricing problem
     - If preventing a vertex in MAPF, delete all paths that use this vertex in the master problem, and prevent the pricer from visiting this vertex
- Method 1 is easier but method 2 is stronger[18]

[18] M.E. Lübbecke, J. Desrosiers, Selected topics in column generation, Operations Research, 53(6):1007-1023, 2005.

Simple MAPF Branching

- Want to do this:
  1. Select a vertex v used fractionally by an agent a
  2. Impose
  $$\sum_{p \in P_a} x_v^p \lambda_p \leq 0 \qquad \text{or} \qquad \sum_{p \in P_a} x_v^p \lambda_p \geq 1$$
  3. Hard to implement the dual multiplier of $\sum_{p \in P_a} x_v^p \lambda_p \geq 1$ in A*
- For today, implement a weaker branching rule:
  1. Select two locations l_1, l_2 in use by one agent a at the same time t
  2. Add the constraints
  $$\sum_{p \in P_a} x^p_{(l_1,t)} \lambda_p \leq 0 \qquad \text{or} \qquad \sum_{p \in P_a} x^p_{(l_2,t)} \lambda_p \leq 0$$
  3. Does not partition the solution space, i.e., the same solution can be in both subtrees
  4. These constraints look like vertex conflicts, so we know how to handle their dual multipliers

Summary

1. Defined a branching rule for splitting a branch-and-bound node into children nodes in which none contain the fractional solution of the parent
2. Described two methods to action a branching decision
   - Add a row in the master problem and let its dual multiplier handle it
   - Delete all incompatible plans in the master problem and prevent their regeneration in the pricing problem
3. Showed how adding a branching row can impact the pricing problem via its dual variable

Task 4

1. The code implements the weak but simple branching rule

$$\sum_{p \in P_a} x^p_{(l_1,t)} \lambda_p \leq 0 \qquad \text{or} \qquad \sum_{p \in P_a} x^p_{(l_2,t)} \lambda_p \leq 0$$

2. `[TASK4a]` in `vertex_branching.py`
   - Read this code
   - It selects two locations l_1, l_2 in use by an agent a at a common time t
   - Then it adds a local constraint (above) to each of the two children nodes preventing the use of the vertex
3. `[TASK4b]` in `pricer.py`
   - Read this code
   - It imposes the dual multiplier from branching constraints

Conclusion

Summary

- Multi-agent path finding problem
  - A fundamental problem in AI planning
- Single-agent path finding problem
  - Search algorithms from AI planning
- Integer programming
  - Solving discrete optimisation problems via a sequence of continuous relaxations in a branch-and-bound search tree
- Exponential-size models
  - Stronger lower bound
  - Eliminate symmetry
  - Exploit substructure
  - Richer geometry
  - Huge number of variables and constraints
- Column generation
  - Exploit substructure
  - Use specialised algorithms to find improving plans/paths/schedules
- Row generation
  - Check for violated constraints and generate them on-demand
- Branching
  - Achieve integrality in the branch-and-bound tree
  - Two methods for imposing branching decisions compatible with column generation

Solving Exponential-Size Integer Programs Using Branch-and-Cut-and-Price

Edward Lam
edward.lam@monash.edu
ed-lam.com